2

FINAL TECHNICAL REPORT

The Design of Griffin: A Common Prototyping Language

R.B.K Dewar, B. Goldberg, M.C.Harrison,
E. Schonberg, D. Shasha

Department of Computer Science
Courant Insitute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

The Design of Griffin: A Common Prototyping Language
Final Technical Report

Principal Investigators: Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, Dennis Shasha
PI Institution: New York University
PI Phone Number: (212) 998-3495 (B. Goldberg)
PI E-mail Address: griffin@cs.nyu.edu

Grant or Contract Title: Griffin: A Common Prototyping Language
Grant or Contract Number: DARPA/ONR #N00014-90-J-1110
Period of Contract: 1 Oct 89 - 31 Dec 90

1 Productivity measures

- Refereed papers submitted but not yet published: 5
- Refereed papers published: 2
- Unrefereed reports and articles: 2
- Books or parts thereof submitted but not yet published: 0
- Books or parts thereof published: 0
- Patents filed but not yet granted: 0
- Patents granted: 0
- Invited presentations: 3
- Contributed presentations: 3
- Honors received: 2
- Prizes or awards received (Nobel, Japan, Turing, etc.): 0
- Promotions obtained: 1
- Graduate students supported: 4
- Post-docs supported: 1
- Minorities supported: 0

Statement A per telecon
Dr.Andre Van Tilborg Code 1133
Arlington, Va 22217-5000

NWW 6/10/92

The Design of Griffin: A Common Prototyping Language
Final Technical Report

Principal Investigators:	Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, Dennis Shasha
PI Institution:	New York University
PI Phone Number:	(212) 998-3495 (B. Goldberg)
PI E-mail Address:	griffin@cs.nyu.edu
Grant or Contract Title:	Griffin: A Common Prototyping Language
Grant or Contract Number:	DARPA/ONR #N00014-90-J-1110
Period of Contract:	1 Oct 89 - 31 Dec 90

2 Detailed summary of technical progress

This project was the first phase of the development of a language for prototyping large software systems, especially those that will ultimately be implemented in Ada. The Department of Defense has made a commitment to the notion of rapid prototyping as a critical phase in software development that will result in more useful and reliable software systems and lower software maintenance costs. The success and cost-effectiveness of protyping depends on, among other things, a prototyping language that demonstrates expressiveness, flexibility, and conciseness. Griffin, the language being designed in this project, is intended to meet this need. The Griffin language, in subsequent phases of DARPA's Software Prototyping Technology program, will continue to be refined and evaluated by the Griffin project at NYU and Raytheon Corporatation.

The Griffin project's approach to language design consists of integrating important language concepts exhibited in various modern programming languages, together with innovation in a number of areas important to large prototyping systems. Griffin features higher-order polymorphic functions, strong typing with type inference, a general tasking and communication facility, an object oriented programming facility, powerful datatypes (such as sets), and support for persistence and transaction based programming.

The design of the type model, persistence, object-oriented programming support, concurrency and tasking, and high-level datatypes is virtually complete. Consideration of real-time language support has been postponed to a later stage. Current efforts are concentrated on integrating the language features into a coherent, robust language design. In the following sections, we provide a brief technical description of some important aspects of Griffin, namely types, persistence, and concurrency.

2.1 Polymorphic strong typing

In principle, Griffin is an explicitly typed polymorphic language. In practice, however, Griffin will use type inference as much as possible to reduce the amount of information that the user needs to type or needs to change to modify the program.

Griffin distinguishes between *values*, *types*, and *type sets*. Values can be passed to procedures, returned from them, assigned to variables, stored in pointer structures, etc. Every value belongs to exactly one type. Types are second-class in that they are subject to only some of the operations applicable to values. For example, a type can never be the result of a conditional clause. As the name implies, a typeset denotes a set of types. Typesets can hardly be manipulated at all. The precise nature of how values, types, and typesets are used is detailed below.

2.1.1 Type declarations

Each computed *value* in Griffin has a unique *type*. Griffin provides a rich set of pre-defined types, including sets, bags, sequences, unions, functions, and procedures. It also provides a powerful type definition facility for defining new object, record, structure and task types.

Griffin unifies the notions of record, object, structure, and task in the following way. Each such value computed is called an *object*. An object is a fixed set of named values, some of which may be private and some of which may be public. An object can be specified in three parts: its interface, which includes all public components; its private components; and its implementation, which includes the code used to implement public and private operations. A task is an object which contains a body that executes concurrently with the creator of the object.

In explicitly typed Griffin, the introduction of each user-defined identifier will be followed by a colon and a type expression. Value identifiers may denote constants, variables, or parameters:

```
<value declaration> ::= name : <type expression> = expr;
                          -- constant
                        VAR name : <type expression>;
                          -- variable or parameter
                        VAR name : <type expression> := value;
                          -- initialized variable
```

Any expression in Griffin can also be followed by a colon and a `<type expression>`.

Type identifiers are declared as follows:

```
TYPE name = <type expression>;
             -- type abbreviation
TYPE name;
             -- type variable or type parameter
```

An object is a set of named value components, each of which may be public or private. What are called records in Pascal are objects with public variable components. Abstract data types are objects with private data components and public procedure components, and tasks are objects with public procedure components (for communication) and a special procedure component called *body* which executes concurrently with its creator.

An object type can be specified in a single declaration, or can be constructed incrementally by extension of previous object types. The syntax of object declarations is:

```
<typexpr> ::=         { <component-def> ... }
                      <name> with { <component-def> ... }
<component-def> ::=   <pp> <cv> <name> : <typexpr>  <opt-value>
<pp> ::=              public   |  private   |  implementation
<cv> ::=              <empty>  |  VAR
<opt-value> ::=       <empty>  |  = <expr>
```

Abstract data-types will usually be defined by first giving a name to a definition which includes the public components, and then extending this by giving the private components and the bodies the functions and procedures.

2.2 Polymorphism

Griffin provides polymorphic values in three ways: parametric polymorphism, as in ML; overloading, as in Ada; and dynamically dispatched methods (subtyping), as in object-oriented languages. The type of such a value is specified using type variables with *(type) constraints.* A single constraint can take on one of the following forms:

```
<type expression>: <typeset>
<expr> : <type expression>
<type expression> = <type expression>
<type expression> <= <type expression>
```

A typeset describes a set of types, and is typically used as a constraint on the types of parameters of polymorphic functions. For example,

```
TYPE NUMBER = { int, real };
sort [t: TYPE | t: NUMBER] (x: set[t]): seq[t] = ...
```

specifies that `sort` can only be called with either sets of reals or sets of integers.

Overloaded functions can be used by employing typing constraints.

```
sort [t: TYPE | gt: t * t -> bool] (x: set[t]): seq[t] = ...
printset [t: TYPE | print: t -> void] (x: set[t]): void = ...
```

Here `sort` is a routine that can be called with any type for which there exists (in the environment in which `sort` is elaborated) a routine `gt` of type `t * t -> bool`. Similarly, `printset` can be applied to a set of any type for which `print` is defined.

2.3 Persistence in Griffin

The basic rationale for any persistence mechanism is that objects should be preserved beyond the execution of a tool that creates or manipulates them. The persistence of an object should be independent of its type. A persistent object manager must be capable of storing and retrieving arbitrary software objects (e.g. parse trees, symbol tables, abstract syntax graphs, source code, test sets, and bug reports) for a broad spectrum of tools.

Since objects, including graph objects, are in persistent store in a way that is isomorphic to the way they are in memory, tools need not transform an object in any way (e.g. flatten them to byte sequences) to make it persistent. Eliminating this transformation is one of the main motivations for including persistence in the language as opposed to using either a file system or a relational database system to be the persistent store. A second main motivation was to allow users to modify persistent store in a disciplined way, i.e. by using atomic transactions.

A third motivation was to make program identifiers transparently refer to objects in persistent storage. This leaves only one set of names that the programmer need remember. Having separate persistent ames is a reasonable decision for models of persistence that must coexist with different languages, but has no advantage in a monolingual environment.

In fact, a Griffin programmer can start with the following simplified view of the persistence mechanism: there is an eternal program that always runs but is never seen. Any visible program is an expression in this eternal program. The global variables of any visible program are therefore global or top-level variables of the eternal program. Since the eternal program goes on forever, those global variables are still accessible after the visible program disappears. Thus, the programmer can think of persistent store just as a set of identifier mappings that can be included in the current scope.

The design enriches this simple picture with three facilities:

1. Because of the transaction notion, a visible program can declare a set of updates to be all-or-nothing and can restrict uncontrolled sharing.
2. Top-level variables may go into one of an arbitrary number of possible "namespaces" instead of one global one (as characterized by a single eternal program).
3. Persistent objects can be deleted (something that never happens to global variables of an eternal program).

2.4 Generalized Tasking and Communication Facility

The Griffin tasking facility is a generalization of the Ada task model, providing asynchronous and anonymous communication, first class tasks, and a general task type that can be instantiated with different bodies.

2.5 Concurrency

Concurrency in Griffin is a generalization of Ada tasking in the following ways.

- Tasks are objects with independent threads of control. Like other objects, they can be inherited, and descendant objects inherit the interface of their parents.
- Tasks types, like other types, can be parametrized. This allows tasks to have access to their own (externally determined) identity at the time of creation, such as their index in an array, or a printable name (their identity is available as *self*).
- Task types are independent of bodies. The type of a task is determined solely by its interface.
- Tasks communicate by means of channels, which are concurrent-access data-structures with multiple readers and writers. Channels provide symmetric anonymous communication between multiple servers and multiple clients. A task can send a message to a channel (akin to an entry call), and can retrieve a message from a channel, by means of an accept statement. Channels are first-class values, so that channel values can be assigned to channel variables, and passed as parameters.
- Channels may be synchronous (i.e. blocking) or asynchronous: if synchronous, sending a message will require a rendezvous with a task which is doing an accept on this channel; if asynchronous, the message will be copied into the channel, and the sender task will continue execution. Thus channels generalize the rendezvous mechanism on one hand, and message-passing systems via mailboxes on the other.

2.5.1 Task types

The declaration of a task type has the form:

```
type tt = task {channel_declarations procedure_declarations};
```

If a task *T* includes a channel declaration for a channel *C*, then other tasks can communicate with *T* by sending to *C*; other tasks can also accept messages from channel *C*, which is thus a broadcast medium. If the channel declaration is private, then *T* may declare access procedures that allow tasks to access the private channels in restricted fashion (for example, by allowing sends but not receives, or vice versa). Private synchronous channels with public send procedures are equivalent to Ada entries.

2.5.2 Task bodies

All tasks have a component named body, which is a procedure type. There are two kinds of bodies and two corresponding kinds of tasks. The first kind of body contains a non-empty sequence of statements, and specifies the code executed by the thread of control to which the task is bound at the time of creation.

The second type of body is an empty sequence, and denotes a task that need not be bound to a separate thread. Such a task serves as a monitor for its components, that is to say, access to this task guarantees mutual exclusion. Mutual exclusion is not enforced on variables that are not of a task type and that are used without explicit synchronization by several tasks. The empty body of a monitor is simply a convenient abbreviation: the body of such a task is in fact a select statement with unguarded accepts that read or write any data component of the task.

The Design of Griffin: A Common Prototyping Language
Final Technical Report

Principal Investigators:	Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, Dennis Shasha
PI Institution:	New York University
PI Phone Number:	(212) 998-3495 (B. Goldberg)
PI E-mail Address:	griffin@cs.nyu.edu
Grant or Contract Title:	Griffin: A Common Prototyping Language
Grant or Contract Number:	DARPA/ONR #N00014-90-J-1110
Period of Contract:	1 Oct 89 - 31 Dec 90

3 Publications, presentations, reports, and awards/honors

- "Generators and the Replicator Control Structure in the Parallel Environment of ALLOY", by T. Mitsolidis and M. Harrison. SIGPLAN'90 Conference on the Design and Implementation of Programming Languages, White Plains, June 1990.
- "The complexity of type inference for higher-order typed lambda calculi", by F. Henglein and H. Mairson, ACM Symposium on Principles of Programming Languages, January 1991.
- "Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations", by B. Goldberg and Y.G. Park. Proceedings of the 1990 European Symposium on Programming, May 1990. Springer-Verlag LNCS 432, pp. 152-160.
- "Shared Variables and Ada 9X Issues", by Robert B.K. Dewar. CMU Software Engineering Institute Special Report SEI-90-SR1.
- "The Fixed-Point Facility in Ada", by Robert B.K. Dewar. CMU Software Engineering Institute Special Report SEI-90-SR2.
- "Beyond Fail-Stop: wait-free serializability and resilience in the presence of slow-down failures," by Dennis Shasha and John Turek. Submitted to the ACM Symposium on Principles of Database Systems.
- Henglein, F., "Type Inference with Polymorphic Recursion", submitted to ACM Transactions on Programming Languages and Systems, July 1990, under review;
- Cai, J., Facon, P., Henglein, F., Paige, R., Schonberg, E., "Program Transformation and Data Structure Selection", submitted to IFIP TC 2.1 Working Conference, to be held May 1991, under review;
- "Escape Analysis on Lists: Optimizing Storage Allocation and Reclamation in Higher Order Functional Languages", by Y.G. Park and B. Goldberg, Submitted to the 1991 ACM Symposium on Principles of Programming Languages.
- "Persistent Linda: a tool for parallel programming providing transactions," by Brian Anderson and Dennis Shasha, in preparation.
- "Tag-Free Garbage Collection for Typed Programming Languages", by B. Goldberg. In preparation.

The Design of Griffin: A Common Prototyping Language
Final Technical Report

Principal Investigators:	Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, Dennis Shasha
PI Institution:	New York University
PI Phone Number:	(212) 998-3495 (B. Goldberg)
PI E-mail Address:	griffin@cs.nyu.edu
Grant or Contract Title:	Griffin: A Common Prototyping Language
Grant or Contract Number:	DARPA/ONR #N00014-90-J-1110
Period of Contract:	1 Oct 89 - 31 Dec 90

4 Transitions and DoD interactions

The technology transfer in the project has been both incoming and outgoing. In order to understand the needs and problems of the Ada programming community, the project has established a dialogue with the Raytheon company, through their software development division. Raytheon software developers and managers have visited NYU for presentations and in-depth discussions. In addition, the Griffin project has been in regular contact with other research groups involved in Darpa's CPL program. Such groups include Yale/Software Options and Duke/Kestrel.

So far, the outgoing technology transfer has mainly been to the ADA-9X effort. Robert Dewar, a member of the Griffin project, is a distinguished reviewer for ADA-9X and has presented ideas developed by the Griffin project for consideration in ADA-9X. In particular, Griffin's generalized tasking and communication facility is of particular interest for ADA-9X.

The Design of Griffin: A Common Prototyping Language
Final Technical Report

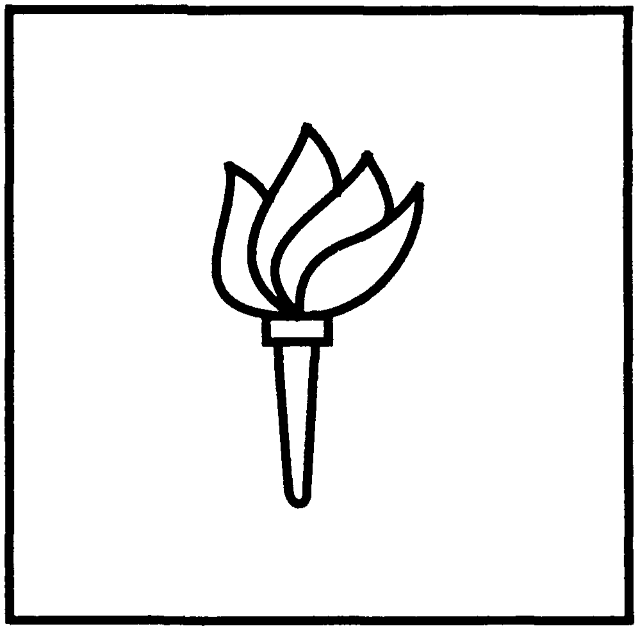
Principal Investigators:	Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, Dennis Shasha
PI Institution:	New York University
PI Phone Number:	(212) 998-3495 (B. Goldberg)
PI E-mail Address:	griffin@cs.nyu.edu
Grant or Contract Title:	Griffin: A Common Prototyping Language
Grant or Contract Number:	DARPA/ONR #N00014-90-J-1110
Period of Contract:	1 Oct 89 - 31 Dec 90

5 Software and hardware prototypes

The Griffin project is still in the language design phase. Thus no prototype implementation has been constructed yet. In the next phase of the project, a prototype language implementation is planned in order to experiment with, and further refine, the language. In subsequent phases, a production quality implementation of the Griffin language is planned. This will require collaboration with an industrial partner.

GRIFFIN

Preliminary Language Reference Manual



CONTENTS

CHAPTER 1: Introduction

This is still a working document. It is not a tutorial. It is not a formal definition. It is not a rationale, though the motivation for some features is explained more than others.

This version, June 1992, is more complete than the November 1991 version. All of the significant features of the language are described, some with more precision than others. With respect to syntax, this document reflects our general approach, but we anticipate that details of the syntax may still change as we gain more experience with the language.

Included for our own purposes is an Appendix containing the actual syntax used by our parser.

The main changes are in the following areas:

- arrays and sequences have been unified; sequences are eliminated;
- the syntax for ADTs has been changed to allow non-record representations;
- iterators and aggregators have been spelled out in more detail;
- the keyword **class** has been added, and polymorphism and unions are described in such terms;
- types have been added to the pattern-matching facility, to provide conformity tests;
- the Ada interoperability chapter has been extended;
- the persistence chapter has been brought into line with our implementation plans;
- the visibility chapter has been changed to eliminate the required nesting;

2

CHAPTER 2: Lexical and Syntactic Structure

In this chapter, we describe the low-level lexical structure for Griffin based on an ASCII character representation. Griffin uses many ideas from Haskell [huwa90], and some from ML [ap88] and Algol 68 [wmpk69].

2.1 Notational Conventions

The following notational conventions are used for program examples and syntax descriptions:

[*pattern*]	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
pat_1 | *pat_2*	choice
pat_{{pat'}}	difference --- elements generated by *pat* except those generated by *pat'*
keyword	terminal syntax for keywords
terminal	terminal syntax for other terminals

Throughout this manual, BNF-style syntax is used to describe the language. Productions have the form:

nonterminal ::= *alt_1* | *alt_2* | ... | *alt_n*

Griffin program fragments may contain "holes" which are written in italics, as in

```
if e_1 then e_2 else e_3
```

The names of "holes" will typically be mnemonic, e. g. *e/* for expressions, *d/* for declarations, *t/* for types, etc. In order to avoid potential ambiguities, meta-syntax such as | and [...] is indicated by using roman font, whereas concrete terminal syntax such as `|` and `[...]` is indicated by using typewriter font. Comments in the syntax are preceded by two hyphens.

Although Griffin source programs are currently written using an ASCII character representation, future developments will address broader character standards.

2.2 Lexical Namespaces

There are four kind of names in Griffin: those for *variables*, *constants*, and *constructors* denote values; those for *type identifiers*, *type variables*, and *type constructors*, and those for *signatures* refer to entities related to the type system; and those for *namespaces/* and *directories/* refer to persistent storage repositories. Griffin is case-insensitive. However, many of the examples are written with signature names beginning with an upper-case letter.

2.3 Syntactic Principles

In Griffin, the syntax for *entities/* resembles the syntax for the corresponding *type expressions/*. For example, a tuple value is written as `(2,3.7)`, and its type is written as `(int,real)`. Furthermore, all *bindings/* have the same form:

def_class identifier [*sep specifier*] = *expression*

For example, the type of functions from `int` to `int` could be defined as

```
type int_to_int = int->int
```

2.4 Keywords

Keywords in Griffin are not reserved words, and are distinguished from identifiers in some unspecified way. The way in which such keywords are entered from a keyboard is not specified here. In this manual keywords are printed in bold type, and most use mnemonics of few letters. The particular spellings of these keywords in printed or displayed programs may change.

2.5 Layout and Indentation

In order to encourage a uniform layout and indentation style, Griffin provides a convenient layout mechanism in all syntactic constructs, which is generalized from the mechanism used by Haskell. Lists of declarations and sequences may occur at various positions in Griffin programs, e. g. in the **then** part of an **if** statement. For example, the syntax of an **if** statement is:

```
if e_1
then stmt_1 ; stmt_2 ; ... ; stmt_n ;
[ else stmt_{n+1} ; stmt_{n+2} ; ... ; stmt_{n+m}  ]
end if
```

Griffin permits the omission of the terminators by using *layout* to express the same grouping information. This allows both layout-sensitive and -insensitive styles of coding within one program. Using a layout-insensitive coding style, Griffin programs can be generated mechanically by other programs. The layout (or "off-side") rule is applied whenever indentation follows a keyword in a construct expecting a list. Then, the indentation of the next lexeme is remembered. For each subsequent line, if it contains only whitespace or is indented more, then the previous declaration or statement is continued; if it is indented the same amount, then a semicolon is inserted and a new declaration or statement begins; and if it is indented less, then the list ends and the appropriate terminator is inserted.

2.6 Comments

Griffin has two kinds of comments:

- inline comments extending up to the end of an input line starting at the *inline comment symbol* --
- comments enclosed between an *open comment symbol* (* and a *close comment symbol* *)

Enclosed comments can be used to implement *annotations*, i. e., comments at the syntactic rather than at the lexical level. Any syntactic unit in Griffin can be annotated by a piece of text that is then part of that unit. In this way the documentation process can be supported by a syntax-directed editor. Griffin annotations will be written as enclosed comments but have fields in the comment text that indicate their connection with certain syntactic units.

2.7 Lexical Program Structure

```
program       ::= { lexeme | whitespace }
lexeme        ::= varid | conid | varop | conop | literal | special | operator | keyword
literal       ::= integer | real | char | string
special       ::= ( | ) | , | ; | [ | ] | _ | { | }
whitespace    ::= whitestuff {whitestuff}
whitestuff    ::= newline | space | tab | vtab | formfeed | comment | ncomment
newline       ::=  a newline (system dependent)
space         ::=  a space
tab           ::= a horizontal tab
vtab          ::=  a vertical tab
formfeed      ::=  a form feed
tab           ::= a horizontal tab
comment       ::= -- {any} newline
ncomment      ::=  (* { whitespace | any } *)
```

```
any         ::= graphic | space | tab
graphic     ::= large | small | digit | ! | {\tt "} | # | $
small       ::= a | b | ...| z
large       ::= A | B | ... | Z
digit       ::= 0 | 1 | ... | 9
```

2.8 Identifiers and Operators

```
symbol      ::= ! | # | $ | % | & | * | + | . | / | < | = | > | ? | @ |\| ^ | | | ~
```

Griffin distinguishes between *alphanumeric\/* and *symbolic\/* identifiers. An alphnumeric identifier consists of a letter followed by zero or more letters, digits, underscores, or primes. A symbolic identifier is formed from one or more symbols, as defined above.

2.9 Numeric Literals

```
integer ::=    digit{digit}
real    ::=    integer.integer |
               integer[.integer]\(e|E\)[-]integer
```

There are two kinds of numeric literals: integers and reals. A real literal must contain a decimal point with digits before and after the decimal point, and an optional exponent. When an exponent is given, no fractional part needs to be given. Negative numeric literals are obtained by application of unary - (see Chapter [Ref: expressions]).

2.10 Character and String Literals

Character literals are written between single quotes, as in `'a'`, and string literals between double quotes, as in `"Good morning"`. Various formatting codes may be used in character and string literals. Long strings may extend over several lines by enclosing one line break and optional whitespace between two backslashes, as in

```
"This is a string that \
     \extends over two lines"
```

(Long strings containing backslashes can be written as long as the backslash is not the last non-whitespece character on one line and the first on the next line).[1]

[1] Why not just concatenate shorter strings – MCH.

CHAPTER 3: Types

3.1 Overview and Rationale

In principle, Griffin is an explicitly typed polymorphic language [Milner78, Reynolds85]. In practice, however, Griffin will use type inference as much as possible to reduce the amount of information that the user needs to specify in a program (cf. [Suzuki81, Borning82, Mishra85, Johnson86, Mitchell88, Graver90]). Which parts of the type information may be elided such that they can feasibly and unambiguously be reconstructed by a type inference system is not specified in this report.

This chapter describes Griffin's explicit type formalism, i.e. the form a program takes when all type information is made explicit. (Note that since Griffin is a polymorphic language, it is possible to write functions and procedures whose arguments and variables have types containing variables. Discussion of polymorphic types is postponed to a later chapter).

3.2 Type expressions

Each **value** in Griffin has a unique **type**. The introduction of each user-defined value **identifier** is annotated with a type expression *typexpr*. Value identifiers denote constants, variables, or parameters:

```
value_declaration ::=
        names : typexpr = expression ;
                -- constant
  |     var names : typexpr ;
                -- variable or parameter
  |     var names : typexpr := expression ;
                -- initialized variable or parameter with default value

names ::=
        identifier
  |     identifier , names
```

Any expression in Griffin can be annotated by following it with a colon and a *typexpr*. Type expressions are constructed out of primitive types (such as `int`), built-in type constructors (such as set), user-defined type names, user-defined type constructors (parameterized types), and type variables. Parameterized types and type expressions containing type variables (polymorphic types) are discussed in separate chapters.

3.2.1 Type declarations

Types are introduced by means of type declarations:

```
type_declaration ::=
        type identifier = typexpr ;                  -- (constant) type definition
  |     type identifier = new typexpr ;              -- name equivalence

typexpr ::=
        int | real | char | bool | atom | void       -- primitive types
  |     composite_type_declaration
  |     fun (formal_typexpr, ...) typexpr            -- function types
  |     proc (formal_typexpr, ...)                   -- procedure types
  |     tuple_declaration
  |     record_declaration
  |     task_specification
  |     channel_declaration
|       ptr typexpr                                    -- pointers
|       enum { identifier, ... }                       -- enumerated types
|       alt { member, member, ... }                    -- algebraic/data types
|       type_name                                      -- named type
|       type_con_name "[" actual_typexpr, ... "]"      -- parameterized type

formal_typexpr ::=
        [ mode ] typexpr

mode ::=
        in | out | inout

actual_typexpr ::=
        typexpr
|       constant

type_name ::=
        qualification name

member ::=                                             -- member of data type
        identifier                                     -- nullary member
|       [ identifier of ] typexpr                      -- unary member

qualification ::=
        empty                                          -- empty
|       qualification package_name
```

Alternative forms for procedure and function types are:

```
function_typexpr ::=
        formal_typexpr * formal_typexpr ... -> typexpr
|       formal_typexpr * formal_typexpr ... -> void

procedure_typexpr ::=
        formal_typexpr * formal_typexpr ...
```

Polymorphic function types are defined in the next chapter and parameter passing modes are discussed in the chapter of subprograms.

3.2.2 Forward declarations

Types must be declared before their use. Because it is not always possible to give the full definition of a type before its use, Griffin allows forward declarations, which are used to establish an identifier as a type, and indicate that a full definition is forthcoming. The syntax is as follows:

```
forward_type_declaration :=
        type identifier, ... ;
```

3.2.3 Invalid Types

A type is invalid if all values of that type are infinite in size. Variables of such a type cannot be constructed. In addition, for a similar reason, recursive types constructed through records or arrays must use explicit pointers. This is not necessary for recursive types constructed through alt. For example:

```
type bad1 = rec { data : bad1; };                   -- invalid type
type good1 = rec { data : ptr good1; };
type bad2 = arr [int] bad2;                         -- invalid type
type good2 = arr [int] ptr good2;
type good3 = alt { null, a of arr [int] good3, r of rec { data : good3 } .
```

Types **bad1** and **bad2** are invalid for the reasons described above. Types **good1** and **good2** show how to alter the bad types to make them valid. good3 demonstrates that types using alt need not use explicit pointers for recursion.

3.2.4 Examples

The following are simple object declarations:

```
var i : int;
var a : arr [1..10:int] real;
s : bag[int] = {1,2,3};
var s : bag[bag[char];
b : arr [1..n:int] char;
type u;
type t = rec {
   var i : int;
   var p : ptr u;
};
type u = arr [int] ptr t;
type tt = alt { leaf of int, internal of arr [int] tt };
```

The declaration of b, with parameter n, is an instantiation of a parameterized type which can be used either to specify the type of a function parameter by pattern matching, where n is bound in conformance with the actual argument (see section on pattern matching), or size n where n is evaluated during elaboration of the declaration of b.

The declarations t and u describe one way to implement an n-ary tree. Note that a forward declaration was necessary because t and u are mutually recursive. tt is another (better) way of describing the same structure.

3.3 Type Equivalence

[1]Our default is structural equivalence for types, but we need name equivalence also:

```
type identifier = new typexpr ;
```

In analogy with Ada derived types, this creates a new type which is not assignment-compatible with any other type.

3.4 Enumerations and Alternatives

Enumerations define ordered sets of symbols. They are similar to Ada's enumerated types.

```
type colors = enum { red, green, blue };
```

As in Ada, enumerated literals can be overloaded.

Alternatives correspond to ML *datatypes* [ap88] and generalize enumeration types. They may be recursive and parameterized (see Section [Ref: parameterizedTypes]). As an example, consider a binary tree of integers:

```
type inttree = alt { empty, leaf of int, node of (inttree, inttree) };
```

Alts are useful for defining functions in a pattern-matching style (see chapters [Ref: expressions] , [Ref: subprograms]).[2]

[1] How about merging this section into 3.2.1 and removing the definition of *type_expression*?

[2] Do we agree on alt? – MCH.

3.5 Subranges and Constraints

Griffin provides subranges and other constraints by allowing redefinition of the assignment operator. This is described in a later chapter.

3.6 Composite types

[1] Composite types define aggregates of values of other types. All the defined components of a composite type have the same type. The component type of a composite type can be another composite type.

```
composite_type_declaration ::=
        seq "[" typexpr "]"                          -- sequence type
|       arr "[" index_definition; ... "]" typexpr    -- array type
|       bag "[" typexpr "]"                          -- bag type
|       map "[" typexpr "]" typexpr                  -- map type of SETL

index_definition ::= range | enumerated_type_name

range    ::= lb .. ub : discrete_type_name
```

3.6.1 Operations on composite types

Values of composite types participate in the operations of assignment, cardinality and equality. In addition they appear in iterators, in quantified expressions, in membership tests, and in aggregates.

3.6.2 Sequences

Sequences are ordered collections of values of a given component type. Sequences are unbounded in size. Elements of a sequence are numbered from zero. The size of a sequence is the largest index for which there is defined component of the sequence. If a component of a sequence is not defined, then an attempt to use its value in a expression raises the predefined exception *undefined.*

3.6.3 Arrays

Arrays are ordered sequences of possibly fixed size, of values of a given component type. Each component is specified by its index value(s). An array object has a defined value for each sequence of index values that can be formed by selecting one value from each index position. The values of elements can be changed.

Example:

```
var av: arr [int] of real;
var af1: arr [1..5] of real;
var af2: arr [0..99] of real;
```

The type of an array is determined solely by the types of its indices and base. The types of **av**, **af1**, and **af2**, therefore, are all the same, because in all three cases, the index type is `int` and the base type is `real`. If the bounds are specified in the declaration, as for af1 and af2, they are part of the variable and cannot be changed. If the bounds are not specified, as for **av**, then they are part of the value, but not part of the variable.. Therefore, the size of **av** can be changed, but the sizes of **af1** and **af2** cannot. To give **av** a value, we can say

```
av := af1;
```

assuming that the elements of **af1** were initialized prior to the assignment.

[1] We need some examples here, this is very bland – MCH

Since **af1** and **af2** are of fixed size, neither of the following assignments are legal:

```
af1 := af2;
af2 := af1;
```

The assignment

```
af1 := av;
```

is legal only if the current size of **av** matches that of **af1**, ie. automatic sliding is done if necessary, as in Ada.

3.6.4 Operations on ~~sequences and~~ arrays

Operations on ~~sequences and~~ arrays include all the operations on composite types. For all these operations, except membership, n-dimensional arrays are equivalent to one-dimensional arrays of arrays of (n-1) dimensions. ~~Sequences and~~ arrays participate in indexing operations and in slices. One-dimensional arrays participate in concatenation. For multidimensional arrays, the cardinality can be applied to each dimension. ~~Sequences and~~ arrays also participate in componentwise assignments and in slice assignments. Insertion and deletion are defined for ~~sequences, but not for~~ arrays.

3.6.5 Lists

Lists are ordered sequences of values of a component type.

3.6.6 Operations on lists

Operations on lists include cons, which can be used constructively, and in a pattern matchinmg sense.

3.6.7 Bags

Bags are unordered sequences of values of a component type. The same value may appear more than once in a bag. The **multiplicity** of a value is a positive integer that indicates the number of occurences of that value in a bag.

3.6.8 Sets

Bags, all of whose components have multiplicity one, are called sets. They have all the properties of the corresponding mathematical notion (unordered collection of values with no duplicates).

3.6.9 Operations on bags and sets

Operations on bags and sets include all the operations on composite types. In addition, they appear in the multiplicity operation, in union and intersection, insertion, deletion and difference. Note that operations are defined differently for bags and sets. There are no mixed operations between bags and sets, except conversion.

3.6.10 Maps

Maps are associative structures that correspond to the mathematical notion of a single-valued relation. Maps are bags whose components are tuples of length two, such that the multiplicity of each component is one and no two components of the map have the same first component. The set of all first components of map components is the domain of the map; the set of all second components is the range of the map. Multi-valued maps can be represented by sets of pairs[1]

3.6.11 Operations on maps

[1] No multiple-valued maps? – MCH

Operations on maps include all the operations on composite types. In addition, maps appear in retrieval operations and pointwise assignments.

3.7 Tuples and Records

Griffin tuples are heterogeneous, ordered collections of value components, where the number of components is fixed and at least 2. Tuple types are denoted as follows:

tuple_declaration ::=
    type_expr * *type_expr* ...

Tuple components may be variable or constant. They are selected using dot notation indicating the position of the component in the tuple (see chapter [Ref: expressions]). Records in Griffin are sets of named components of arbitrary type:

record_declaration ::=
    rec { *rec_component*; ... }

rec_component ::=
    names [: *typexpr*] = *expression*
| **var** *names* [: *typexpr*] [:= *expression*]

Record components may be variable or constant, and initialized or not (in which case they have a default value of undefined). They are accessed using conventional selected component (dot) notation.

The names of the fields are part of the type of a record. Tuples are considered to be a special kind of record without field names. A literal tuple expression may be assigned to a record without explicit conversion (if the order and type of the components are the same).

3.8 Access Types

3.8.1 Atoms

Atom types are provided to permit the construction of recursive data-structures without using pointers, as in Setl. The only operation on atoms is equality. The type name is `atom`, and the constructor is `newat ()`. The declaration:

```
var a: atom;
```

will declare a variable of type atom, and initialize it to a unique atom, and:

```
a := newat();
```

will create a new atom value and assign it to `a`. Note that if atoms are use with maps to create recusive structures, the type information is associated with the map, not the atom.

3.8.2 Pointers

Pointer types are provided to provide low-level Ada-like data-structures, and to provide non-copying assignment and parameter passing. Any type expression can be preceded by **ptr** to create the corresponding pointer type. For example:

```
var p: ptr ttt;
```

will declare a variable of type pointer-to-`ttt`, and:

```
p := new ttt();
```

will allocate space for a new `ttt`. If `ttt` can be initialized, the notation:

```
    p := new ttt(params);
```

can be used. The keyword new is required to specify that the value is a pointer to a `ttt`, rather than a `ttt` itself. If a type is a pointer, as in

```
    type pttt = ptr ttt;
```

then:

```
    ... pttt(params)
```

will allocate a `ttt` and return a pointer to it. Assignment to a pointer does not copy the value pointed at, only the pointer itself. Similarly, passing a pointer parameter does not cause the value pointed at to be copied in or out. Equality is defined for pointers. The only operation on a pointer, apart from assignment and equality, is dereference, which is done explicitly. We adopt the Pascal notation for dereferencing. Thus:

```
    p^
```

will refer to the value pointed to by `p`, and:

```
    p^ := xyz;
```

will change that value. Note that, unlike C, Griffin does not define addition on pointers, and, also unlike C, it is not possible to create a pointer which points to an existing value or variable.

3.8.3 Atoms and Pointers

Griffin views a pointer as a combination of an atom and a (typed) global map from atom to value. Creating a pointer to a type is viewed as creating a new atom, making the appropriate entry in the map, and returning the atom. Dereferencing a pointer is equivalent to using the map to access the value.

3.9 Functions and Procedures

These are described in a separate chapter.

3.10 Tasks and Channels

These are described in a separate chapter.

CHAPTER 4: Abstract Data Types

4.1 Rationale

One of the objectives of the Griffin design is to unify user-defined types with the object-oriented programming style [MacLennan82,Meyer88]. We do this by way of the observation that the OOPS approach consists of the following: the ability to define extensions of previously defined data structures; the ability to process such extensions with the same code as the parent type or types; and the ability to dynamically dispatch common operations if necessary. We generalize these ideas, giving a powerful type system which provides OOPS facilities as a special case. Griffin distinguishes between *values* and *types*. Values should be considered "first-class citizens" and types "second-class citizens" of Griffin. Values can be passed to procedures, returned from them, assigned to variables, stored in pointer structures, *etc*. Every value belongs to exactly one type.

In this chapter we define ADTs and their signatures. Note that many of these mechanisms also apply to packages and task types, although we only use ADTs in most examples. In the next chapter we discuss polymorphism and OOPs-related issues.

4.2 Abstract Data Types and Signatures

Abstract data types (ADTs) are provided in Griffin, using an encapsulation mechanism similar to Ada package declarations. Griffin also provides packages, which allow the definition of more than one type, and which are described in a later section. Unlike packages, an ADT only defines a single type. The interface of an ADT is called its *signature*. The keyword **sig** is used for defining ADT signatures.

```
declaration ::=
        sig sig_name = sig_expr ;
|       type type_name :: sig_expr = adt_definition ;
sig_expr ::=
        sig_name
|       { typexpr | type_constraints }
|       { type_constraints }                  -- using mytype
adt_definition ::=
        typexpr with { component_definitions }
|       typexpr
type_constraint ::=
        names : typexpr
|       type_name :: sig_expr
component_definition::=
        value_declaration
```

Only components specified in the signature are accessible from outside an ADT. A simple example of an ADT is:

```
sig Point = { p |
   move: (p,int,int) -> void;
   draw: p -> void;
   };
type point::Point = rec{x, y: int := 0} with {
   fun move(p:point; mx,my:int) =>
      begin p.x := mx;   p.y := my; end;
   fun draw(p:point) => drawpoint(p.x,p.y);
   };
```

This defines an ADT with signature `Point` and type `point`. The definition of the signature `Point` should be read:

> A type `p` defined to have the signature `Point` will have operations **`move`** and **`draw`** with types `(p,int,int)->void` and `p->void` respectively.

The definition of the type `point` specifies the representation type (before **with**), and the implementation of the required operations (after **with**). Inside the definition of these operations the details of the representation are accessible.The only operations allowed on type `point` are `move` and `draw`. Examples of use:

```
let
   var p:point;
in
   ... move(p,1,2) ...
   ... draw(p) ...
end;
```

Note that the default assignment operation for ADTs has copy semantics. If pointer semantics is required, as in many object-oriented programming styles, the assignment operator can be redefined (see below), or the user can declare constants and variables to be **ptr** types. The type name of an ADT, if defined as a function, will be interpreted as an initializer for the ADT.

In addition to the operations required to be defined for an ADT by its signature, other functions and procedures which take ADTs as parameters may be defined in the usual way. However, such functions and proceduresare not regarded as part of the type; mechanisms are available for doing this, discussed in the next chapter.

4.2.1 Self-reference in signature definitions

(DISPUTED) Declarations of ADTs can also refer to themselves by means of the keyword **mytype**.These are included to facilitate reuse of existing code beyond that provided by the built-in inheritance mechanisms, discussed in the next chapter.

Examples of such declarations are:

```
sig Printable = {print: mytype->void;};
type eg :: Printable = rec{name: seq[char];} with {
   fun print(x:mytype) => ...    -- do something on x.name
   };
```

4.2.2 Quantification in signature definitions

For most cases, the use of the specified bound type variable (or **mytype**) is powerful enough to define signatures. However, in some cases we may need to introduce other local type variables. We use the convention that in a signature, any identifier followed by a double colon and a type constraint is a local type variable. For example, to define a type which is a pair of identical printable types, we can write:

```
sig Printablepair = { (t::Printable, t) };
```

or:

```
sig Printablepair = { (t,t) | t::Printable; }
```

Every implementation of `Printablepair` must be a pair of identical types with a `print` function defined. Other examples of this notation are:

```
sig Printableset = { set[t::Printable] };
sig Oddpair = { (t1::Printable, t2::Point) };
sig PtrObj = { ptr p | ff : ptr p -> int; ... };
```

Thus, the syntax rule for *typexpr* is extended by:[1]

```
typexpr ::=
        typexpr :: sig_expr
```

This notation, of a *typexpr* followed by a double colon and a *sig_expr*, is used in several other contexts, with somewhat different meanings. These are described in later sections.

4.2.3 Parameterized Types and Signatures

Parameterized types and signatures [Meyer86,Kaes88,Stroustrup88] in Griffin are analogous to generic packages in Ada, consisting of templates which can be instantiated with types and values. Instantiation of signatures and types is static, as in Ada's generic units. [2]

```
declaration ::=
        sig sig_con_name [ formal_type_par; ... ] = sig_expr
  |     type type_con_name [ formal_type_par; ... ] = typexpr
  |     type type_con_name [ formal_type_par; ... ] :: sig_expr = adt_definition
formal_type_par ::=
        names : typexpr
  |     names :: sig_expr
  |     type_con_name [ actual_typexpr, ... ]
sig_expr ::=
        sig_name
  |     sig_con_name [ actual_typexpr, ... ]
  |     { component_declarations }
typexpr ::=
        type_con_name [ actual_typexpr, ... ]
```

The first form for *formal_type_par* is for value parameters, and the other forms are for type expressions. Thus a square matrix can be parameterized with the element type and the index type or size:

```
type sqmat [e::type; n:int] = arr [1..n, 1..n]of e;
type sm10 = sqmat[real,10];
```

Note that *sig_con_name* and *type_con_name* are not signatures or types. Only their instances can be used as signatures and types.

4.3 System-defined ADTs

All types and type constructors discussed in the types chapter are actually predefined ADTs or ADT constructors, although many of them, e.g., **int**, **real**, and **arr**, cannot be defined by other types. Common signatures such as **Int**, **Number**, and **Ordered**, are defined in the library to encourage a uniform programming style. Therefore, **int**, **real**, **char**, and **bool** can be considered predefined ADTs, and **ptr**, **seq**, set, **bag**, and **map** are predefined ADT constructors. The constructors **ptr**, **seq**, **set**, and **bag** take one formal type parameter; **arr** and **map** take two. These predefined ADTs all have well defined signatures in the library. Tuples, records, and tasks are special forms of ADTs. Tuples have ordered components to facilitate pattern matching. Records are used to define ADTs with all public components, so the record type has a simpler syntax than general ADTs. Tasks have a **body** component with a function type of **void->void**. Task bodies represent separate control threads, and are described in a separate chapter. [3]

[1] Need to fix up syntax to exclude t::s1::s2 -- EO

[2] Shouldn't we relax this for value parameters – MCH

[3] Signatures of enumerated types and union types should be discussed....

4.4 Definable operators for ADTs

All operators in Griffin may be given interpretations for user-defined ADTs. These include the usual arithmetic and boolean operators, and the assignment and equality operators. When operators are redefined, or used as values, they should be preceded by the keyword **op**. Also redefinable are the (implicit) operators for creating and destroying values.

The oveloading introduced by user defined operators must be resolvable at compile time; it is not involved in dynamic dispatching.

For aggregate types, the implicit operators for aggregate formation and iteration can be defined. The signature of an aggregate type is of the form:

```
sig Aggregate[t] = { a |
    for :         a -> (void -> (bool,t));
    with :        (a,t) -> s;
    };
```

The function **forall** should return a pair consisting of a boolean indicating there are more elements, and the next element. The function `with` should be defined to insert a new element into the aggregate.

Also definable are pattern-matching operators. For any binary constructor, the user can define the corresponding decomposition function. In simple cases, such as a Lisp-like **cons**, this will be deterministic, but in general (e.g. for **append**) it will be non-deterministic, so the user must define closures which can be executed repeatedly to return all combinations. The pattern-matching algorithm will use these to generate all possibilities. The name for the decomposition function corresponding to a constructor is formed by preceding the constructor name with ~. Thus if **cons** is defined as type:

```
(et,list[et])->list[et]
```

the decomposition function will be named **~cons** and will be of type:

```
list[et] -> (void -> (bool, (et,list[et]))).
```

The first element returned by the decomposition function will be true if there is (another) decomposition, while the second will give the decomposition.

4.4.1 Subranges and constraints

Griffin provides *constraints* as a general model of subrange types and invariants in abstract data types. To declare a subrange type or to add run-time checked invariants of an ADT, we can define a new type and redefine the implementation of the := function. The following example shows how the **int** type and its subrange type **1..10** are defined.

```
sig Int = {
    op:= : (out mytype,mytype) -> mytype;
    op+ : (mytype,mytype) -> mytype;
    ...
    };
type int::Int = intrinsic int with {
    fun op:= (x,y) => intrinsic bitcopy(x,y);
    fun op+ (x,y) => ...
    ...
    };
sig Int_subset = Int + {
    op:= : (out mytype,t::Int) -> mytype;
    };
type int_range[lb,ub:int] :: Int_subset = int + {
    fun op:= (x,y) =>
        if y >= lb & y <= ub then intrinsic bitcopy(x,y);
```

```
            else raise range_error;
        };
    var a,b:int;
    var c,d:int_range[1,10];
    var e:int_range[5,15];
```

Note that int_range inherits the implementation of int and adds a new overloaded := function. Thus, a:=b and c:=d use the same assignment operation, which is just a simple bitwise copy defined in type int. Expression c:=a and c:=e will use the definition of := in type int_range because both int and int_range[5,15] are in Int but are not equal to int_range[1,10]. This definition of int_range is included in the Standard Prelude, so it can be used to specify indices for arrays. Arbitrarily complex subranges, such as a prime number type, can be defined in a similar way:

```
    type prime_number::Int_subset = int with {
        fun op:= (x,y) =>
            if y >= 2 & is_prime(y) then intrinsic bitcopy(x,y);
            else raise range_error;
        };
```

Parameter passing has the same range check as assignments. The default definition of the assignment operation is:

```
    op:= : (out mytype, mytype) -> mytype;
```

for the interface and

```
    fun op:= (x:out mytype,y:mytype) = bitcopy(x,y);
```

for the implementation. For convenience, these are included by default in every definition of signatures and abstract data types.

4.5 Signatures as visibility constraints

A *constrained_type* is a type whose interface is constrained by a signature:[1]

> *typexpr* ::=
> 		*constrained_type*
> *constrained_type* ::=
> 		*typexpr* :: *sig_expr*

The *constrained_type* is the same as the *typexpr* except that its signature is constrained to be that specified in the following *sig_expr*.

4.6 Coercions

Griffin in general requires that coercions from one type to another be explicitly specified by the programmer. Thus pointers will not automatically be dereferenced if they appear in positions where pointers are not permitted, and integers will not automatically be coerced to reals in an arithmetic expression (though the use of overloaded operators may allow such expressions to be written). Automatic coercion is allowed in four cases: any type may be coerced to void; any type may be coerced to an appropriate constrained type (but not the reverse); any type may be coerced to a function of no arguments which returns that type; and any type may be coerced to a union over a class containing that type (see the next chapter for information on union types). In each case, the coercion is only done if the uncoerced type would generate a type error, and if the coercion is unique. For example, a function could be overloaded with type t and type void->t; in that case any argument of type t would not be coerced.

[1] Once again, need to exclude t::s1::s2 --EO

4.7 Extensions and Inheritance

Griffin provides the ability to extend both signatures and types, using the inheritance mechanism. Inheritance in Griffin is equivalent to textual copying, subject to changes of function and procedure values (but not types). The syntax is:

```
sig_expr ::=
        sig_expr + sig_expr
adt_definition ::=
        typexpr [ + record_typexpr ] [ with { component_definitions } ]
```

Only types whose implementation is a record type can be extended in this way, and the extended representation type must also be a record. For example:

```
sig circle = point + { setr: (circle,real)->void };
type circle :: circle = point + rec{rr: real} with {
    fun setr(c,r) => c.rr := r;
    fun show(c) => drawcircle(c.xx,c.yy,c.rr);
    };
```

would add two new components, a variable **rr** and a constant procedure **setr**, and redefine **show** to invoke the appropriate procedure. When a signature or ADT is inherited the binding of **mytype** changes accordingly.

4.8 Packages

Packages similar to those in Ada are also provided in Griffin. Packages allow the simultaneous definition of several types, and the implementation of such types can depend on each other. Griffin allows the definition of more than one implementation for a given package, so that several bodies (implementations) can be associated with the same Griffin package specification. Package specifications consist of declarations of (constant and variable) values and objects, types, and possibly other packages, and take the form:

```
declaration ::=
    pkg pkg_spec_name = { declarations } ;
```

Package implementations provide values for entities declared in the package specification, and can also declare additional entities that are only visible within the implementation. Implementations are declared as follows:

```
declaration ::=
    body pkg_body_name :: pkg_spec_name = { declarations } ;
```

As in Ada, package specifications provide interfaces and control visibility, while implementations describe internal organization of visible entities. Details of the implementation are only usable within the package body.

4.8.1 Parameterization and Inheritance of Packages

As signatures and ADTs, package specification and implementation can be parameterized and inherited:

```
declaration ::=
    pkg package_name [ formal_type_par; ... ] = package_spec ;
|   body package_name [ formal_type_par; ... ] = package_body ;
package_spec ::=
    package_name
|   package_name [ actual_typexpr, ... ]
|   [ package_name + ] { declarations }
```

```
    | [ package_name [ actual_typexpr, ... ] + ] { declarations }
package_body ::=
    package_spec
```

4.8.2 Example

As an example, we give the code to implement based sets (see Setl notes).[1]

[1] Need to insert this example -- MCH

CHAPTER 5: Polymorphism and OOPS

Griffin provides bounded polymorphic functions, non-homogeneous aggregates, and dynamic dispatching. All of these are defined in terms of classes, which are sets of types which have a common signature. Thus a bounded polymorphic function is defined over types which are constrained to be in particular classes, a non-homogeneous aggregate can contain elements which are (tagged) types in a class, and dynamic dispatching is done when operations are applied to such elements. The details of how this is done are given in later sections. First we define how classes can be defined in Griffin.

5.1 Classes

In Griffin, the meaning of the word class is similar, but not identical, to its meaning in other languages, such as Eifel, Haskell, C++, and Ada9X. A Griffin class is a set of types which have a common interface. Interfaces are defined by a signature name, or a signature-expression. Class denotations have the following syntax:

```
declaration ::=
    class class_name = class_expr
class_expr ::=
    class_name
    sig_name
    all sig_name
    sig_expr
```

Classes can be derived from signatures in two ways: if specified just by a signature name, the class contains exactly those types which have been declared with that signature name (or extensions of it), or are extensions of such types (see later sections for discussion of extensions). Such classes are usually denoted in practice by the signature name, rather than by the definition of a separate class name (and are sometimes called semi-open type-sets in the literature).

The second way of defining classes is by giving a *class_expr*, or by preceding a signature name by the keyword **all**. A class defined in this way includes all types which have definitions forall operations in the signature. (These are sometimes called open type-sets. In existing strongly typed object-oriented programming languages, only semi-open type sets are available. However, in dynamically typed languages such as Smalltalk, type sets are open).

Classes are not values in Griffin in any sense. There are no operations on them, but they can be named.

5.1.1 Putting types into classes

Sometimes it is desirable to specify that an already-defined type should be regarded as being in a class, even though its definition made no reference to the signature name. For this we use the syntax

```
declaration ::=
    type type_name :: class_name ;
    type type_name :: class_name = type_name ;
```

The first form puts the type in the class, if the type has a suitable interface. The second defines a new type that has the same operations and implementation as the type on the right-hand-side, but is in the specified class. Both forms can be used to incorporate into a type functions and procedures not included in the original type definition.[1]

[1] We need to add a renaming facility here - MCH

5.2 Polymorphism

Griffin provides bounded polymorphic functions. These are functions whose parameters have types which contain type variables, and can therefore be applied to more than one type. These types can also be constrained to belong to particular classes, so the polymorphic function can make use of the functions which are guaranteed to be defined. Type expressions used to specify parameters in polymorphic functions can use the additional form:[1]

```
typexpr ::=
    typexpr :: class_expr
```

to constrain the parameter types. For example:

```
sig printable = {print: mytype -> void};
proc print2 (x : t :: all printable) =>
    begin print(x); print(x); end;
```

would define print2 on all types which had a print function in their signature. Similarly:

```
proc printset (s: set[t::all printable]) =>
    for x in s loop print(x) end loop;
```

would allow printing of any set of printable elements. Note that the same mechanism allows ML-like parametric polymorphism as a special case. For example:

```
sig Any = {};
fun length (s: list[t::all Any]):int =>
    if s = nil  then 0
                else 1 + length(tl(s));
```

allows the use of an unconstrained type variable (defined with the class Any which contains all types). This allows the definition of functions whose only operations on such types are parameter-passing and assignment.

A function or procedure whose parameters have constrained types may only contain references to operations which are permitted by the constraint. For example:

```
sig Ord = {op<: (mytype,mytype)->bool; };
class Ordered = all Ord;
fun sort_set (x:set[t::Ordered]):seq[t] => ...
```

could define a sort function which would work over any set whose element type has a less-than function. To make a function polymorphic with respect to a family of parameterized types, we may write:

```
fun inverse (A:sqmat[e::type,n:int]):sqmat[e,n] =>      ;
```

With parameterized signatures, the sort_set function can now be defined more generally as:

```
fun sort (x:t::Aggr[et::Ordered]): seq[et] => ...
```

which will take any homogeneous aggregate of ordered elements. Signature Aggr[et] can be defined as a parameterized signature with common operations of aggregate types.

The following example shows a function f1 which accepts only t1 and t2. However, function f2 would accept t1, t2, and t3.

```
sig sig1 = ...
sig sig2 = sig1 + ...
sig sig3 = ...-- the same spec as sig1
type t1::sig1 = ...
```

[1] Need to exclude t::s1::s2 again -- EO

```
type t2::sig2 = ...
type t3::sig3 = ...
fun f1(x:t::sig1) => ...
fun f2(x:t::all sig1) => ...
```

5.2.1 Predefined and user-defined types

Since the "built-in" types can be viewed as just predefined ADTs, we can write polymorphic functions which accept both predefined types and user defined types. For example, the following declares a polymorphic function to compute inner product:

```
fun product(x,y:arr[1..n](t::all Int)): t => ...;
```

This function accepts either arrays of system defined integers or any user defined type with signature compatible with Int. If the keyword all is left out, this function can only be appled to types explicitly derived from signature Int.

5.3 Union types and OOPS

Griffin provides tagged union types defined in terms of the corresponding class:

```
typexpr ::=
    union class_expr
```

A union type can be considered to be some value whose type is in the class, together with a (hidden) type-tag. A union type can be used, like other types, to declare constants and variables. For example:

```
sig Printable = { print: mytype -> void; };
type anyprintable = union Printable;
```

defines a union type called **anyprintable** which may contain any type which is derived from the **Printable** signature. Values whose type is in a type-set may be coerced to a union over the same type-set. For example, if the type **chr** is derived from **Printable**, then we can write

```
var p: anyprintable;
var c: chr;
...
p := anyprintable(c);
```

which tags the **c** and assigns it to **p**. Subsequently,

```
print(p);
```

will effect run-time dispatching of the appropriate print function. Similarly,

```
var s: set[anyprintable];
```

would be a heterogeneous set all of whose elements could be printed, by, for example:

```
for p in s loop
    print(p);
    end loop;
```

Similarly, a function which is polymorphic over the same type set can be applied to the corresponding union type. Union types can also appear in pattern-matching operations which can test for the actual type. Note: the expression **union** *sig_name* can be considered to be equivalent to class *type_name* in some other languages.

5.3.1 Extensions and Inheritance

Consider the well-known defect in the old version of the type system of Eiffel [Cook89a], which defines an **eq** function in parent class **P**, and then redefines it in a child class C which has additional components. The problem arises when C redefines **eq** and makes reference to components of C which are not in **P**, so if invoked with one object of (actual) class C and one of (actual) class **P**, it will refer to the non-existent component of the latter. In Griffin we can express the desired (safe) inheritance in a number of ways using ADTs and signatures. If we write:

```
sig SigP = {eq: (mytype,mytype) -> bool; };
type P::SigP = rec{a:int;} with {
    fun eq(x,y) => x.a = y.a;
    };
type C::SigP = P + rec{b:int;} with {
    fun eq(x,y:mytype) => (x.a=y.a) and (x.b=y.b);
    };
```

then **eq** is only required to be defined for arguments of the same implementation subtype, and is not defined on union types over the **SigP** class..

5.3.2 Subtyping

(TENTATIVE)Assignment of a type **xt** to a type **t**, where **xt** inherits from type **t**, is legal only if components in **t** but not in **xt** have explicit default values.

5.3.3 ADT names and signature names

Most OOPS languages are type-oriented rather than signature-oriented, and programmers think of extending types rather than signatures. To facilitate this, Griffin allows ADTs to be defined without explicit signature names. For example:

```
type ccircle = circle + rec{....} with {...};
```

is used to define a new ADT named **ccircle**. Its implementation is an extension of **circle**'s implementation and it may also inherit some definitions from **circle**. Later, if we have to refer to the signature of **ccircle**, perhaps to define a polymorphic function or union type over it, we just write sig **ccircle** where the signature name is expected.

5.4 Package Polymorphism

(TENTATIVE)Griffin also provides polymorphism over packages. A function or procedure may be parameterized with respect to a package, which may be constrained by specifying the package specification. For example, the following code could define a function **sort** which would work on any type named **t** which was defined in a package with specification **p**. Inside this function, all other entities defined in the package specification would be available.

```
fun sort [i::p] (x: set[i.t]): seq[i.t] => ...
```

The details of how this might work are described in the paper by Henglein and Laufer.

CHAPTER 6: Names and Expressions

This chapter describes names, expressions and related constructs in Griffin. Names denote entitities in a program. These can be values, objects, subcomponents of composite objects, types, packages, signatures, tasks, and channels. Names that denote values can appear in expressions and be operands in a computation. Names that denote objects can appear as operands in constructs that have side-effects, such as assignments. Names of types can appear in declarations. The syntax of names corresponds to that of Ada.

name ::=
| *simple_name*
| *indexed_component*
| *selected_component*
| *slice*

6.1 Expressions

The evaluation of an expression yields a value. The expressions of Griffin include most of the expression forms of Ada. In addition, Griffin aggregates generalize the aggregates of Ada and provide a concise notation for set, bag, and tuple denotations, as well as for user-defined types for which explicit constructors are defined. Iterators provide convenient abbreviations for common loop constructs in a functional style. Iterators are used in constructors as well as in quantified boolean expressions. Pattern-matching is used to specify local bindings and to provide a limited form of goal-directed evaluation. Pattern matching is also a source of non-determinism in programs. Pattern-matching is used in iterators, subprogram definitions, and case statements.

expression ::=
| *basic_expression*
| *defined_expression*
| *expression_sequence*
| *aggregate*
| *composite_expression*
| *subprogram_expression*
| *assignment_expression*
| *loop_statement*
| *labelled_expression*
| *task_expression*

Although Griffin is not strictly a functional language, the rich set of expression forms makes it possible to program in Griffin in a functional style. Subprogram expressions are discussed in the chapter on subprograms. Control constructs such as conditionals, cases and iterators can be used in expressions, so that the user can, if he chooses, avoid most imperative constructs. If an imperative style is appropriate, assignments are available, both as expressions and as statement forms. Following other quasi-functional languages, statements are simply expression forms that yield the value void. Wherever an expression appears, a sequence of expressions can be used. The value of a sequence of expressions is the value of the last computed expression in the sequence.

expression_sequence ::= *expression* ; { *expression* ; }

6.2 Pattern Matching and Binding

match ::= *pattern* [| *guard*] => *expression* { *pattern* [| *guard*] => *expression* }
guard ::= *boolean_expression*

A match specifies bindings of names to values, and the evaluation of an expression. A pattern describes the type and structure of a simple or composite value, and introduces one or more names, called the *bound variables* of the pattern, to designate components of this value. Patterns are used in iterators, case statements, and subprogram definitions.

A pattern has an implicit argument, called its subject, which is specified by the enclosing context of the pattern or match.

1. In an iterator, the subject is the current component value of the domain.
2. In a case statement, the subject is the case expression.
3. In a subprogram definition, the subject is the tuple of formal parameters of the subprogram.

The evaluation of a match is said to succeed if the value of the subject has the same structure as that of the pattern, and if the guard has value `true`. If a match succeeds, the bound variables of the pattern become bound to the corresponding components of the subject, and the value of the match is the value of the corresponding expression. If a match consists of several patterns, the match succeeds if any of the patterns matches the value of the subject, and the value is that of the corresponding expression. A pattern lacking an explicit guard has an implicit guard with the value `true`. There is a pattern corresponding to each predefined type and to each user-defined value constructor in the language. Griffin supports non-deterministic pattern matching. In the case of a non-deterministic pattern P with guard G, the match succeeds if there is some instance of P such that it matches the argument and the guard G succeeds.

6.2.1 Bound variables

Each simple name in the pattern introduces (i.e. declares) a new bound variable. The scope of the declaration is at least the accompanying guard and expression. When an iterator appears in a loop then its scope extends to the expressions in the body of the loop. The same name may occur more than once in the pattern, in which case each occurrence must be bound to the same value for a match to succeed. A bound variable cannot be modified by explicit assignment, nor by appearing as an actual corresponding to an in out formal parameter.

6.2.2 Patterns

The patterns corresponding to the predefined types are as follows:

```
pattern ::=

  | atomic_pattern
  | array_pattern
  | bag_pattern
  | tuple_pattern
  | map_pattern
  | identifier as pattern
  | ∈ aggregate_expression
  | pattern : type_expression
```

Patterns for composite types make use of the constructors for the type.

The last alternative for a pattern, namely a pattern followed by its type, can be used in those cases in which the type of the subject can vary (such as union types and exceptions) to perform testing on the type of the subject. In such cases, the types of the matches within a single case statement or parameter list can differ (within the constraints of the possible types of the subject). In most cases, however, the types of the matches must be identical. Pattern-matching using types is covered in section [Ref: pattern matching using types] in this chapter and in chapter [Ref exceptions].

6.2.3 Wild card pattern

The pattern _ (underscore) matches anything. It is not a variable and cannot be used in the expression associated with the pattern.

6.2.4 Atomic patterns

```
atomic_pattern ::= identifier | literal
```

A pattern that is an identifier binds the identifier to the subject. A pattern that is a literal succeeds if the subject has the same value as the literal.

6.2.5 Array patterns

```
array_pattern ::=

|   [ [ pattern  { , pattern } ] ]
|   pattern ^ array_pattern           -- Cons.
|   array_pattern & array_pattern   -- Catenation.
```

An array pattern matches an array. The first form matches its subject if the component patterns match the corresponding components of the subject. If the subject is longer than the pattern, additional components are not considered in the match (that is to say, a pattern can successfully match longer subject). The second form matches a pattern if the first component matches the first component of the subject, and the second matches the rest of the subject. The third form is a non-deterministic pattern: it matches the subject if the subject can be split into two contiguous portions that match the corresponding patterns. Either portion of the subject may be the empty array.

6.2.5.1 Examples

```
-- A filter selects those components of an array that satisfy a given predicate.
fun filter(f,[]) => []
 || filter(f,x^xs)  | f(x) => x ^ filter(f,xs)
 || filter(f,_^xs) => filter(f, xs)
-- A non-deterministic definition of membership:
-- x is a member of an array if it appears anywhere in the array.
fun member(x,[]) => false
 || member(x,_&x&_) => true
```

6.2.6 Set and Bag patterns

Patterns corresponding to sets and bags use either an aggregate or the union operator ∪.

```
bag_pattern ::=
|   { pattern { , pattern  } }
|   bag_pattern ∪ bag_pattern      --- Disjoint union
```

The first form matches if the subject has components (in any order) that match the component patterns. Note that a set pattern examines all possible orderings of the components to find a successful match. The second form matches if the subject can be decomposed into two subsets or bags each of which matches the components patterns. In this case all pairs of subsets of the subject may be examined to find a successful match.

6.2.6.1 Examples

```
-- A non-deterministic description of quicksort: the partitioning step is
-- expressed as a pattern involving unions.
fun psort({}) => []
 || psort(big ∪ {k} ∪ small) | (∀ i ∈ big, i >= k and ∀ j ∈ small, j
<= k)
                         => psort(big) & [k] & psort(small)
```

6.2.7 Map patterns

```
map_pattern ::=
|   { pattern -> pattern { , pattern  -> pattern } }
```

| *map_pattern* ∪ *map_pattern* --- Disjoint union

The map pattern specifies the match of individual pairs in the map. The match succeeds if both an element of the domain and its image under the map successfuly match.

6.2.7.1 Examples

-- A function application that uses a memo technique to compute values once.

```
fun memo_apply((f,ftable as {x->z} ∪ _),x) => (z,(f,ftable))
 || memo_apply((f,ftable), x) => let z = f(x) in (z,(f,ftable ∪ {x->z}))
```

A different formulation of the same memo technique.

```
fun memo_apply((f,ftable),x) =
         case ftable of
           {y->z} ∪ _ | x=y => (z,(f,ftable))
           _ => let z = f(x) in (z,(f,ftable ∪ {x->z}))
```

6.2.8 Record patterns

record_pattern ::=
  identifier
| (*field_name* = *pattern* { ,*field_name* = *pattern* })
| (*pattern* { ,*pattern* })

A record pattern matches if the subject is a record with the same field names, and if the corresponding values of the fields match the component patterns. If the field names of the records are omitted, the values are matched positionally, as for an array pattern. In addition to the patterns defined for the predefined types, patterns can be defined to match elements of an abstract data type, using any defined constructor for the type. This facility is described in chapter [Ref: ADTs] , and is currently in the design stage.

6.2.9 Guards

A guard is a boolean expression. A boolean expression can include quantified expressions, that is to say expressions involving the quantifiers ∀ and ∃ (see below). The scope of bound variables introduced by iterators in guards extends over the body of the guard, except for bound variables in existentially quantified expressions, in which case the scope is that of the enclosing pattern or match (including the expression associated with the pattern).

6.2.9.1 Example

```
fun f b | (∃ i ∈ { 1..10 } | b[i] = 6) => i
 || f c => 0
   -- If the actual in a call to f is an array which has a component equal
 -- to 6, then return the index of the corresponding component,
 -- else return 0.
```

6.2.10 Pattern Matching using Types

There are situations in which the type of the subject of a pattern match may be one of a number possibilities. These situations arise in the following ways:

- The subject is a union type.
- The subject is an exception that has been raised and the patterns occur in an exception handler (see chapter [Ref exceptions]).

- The declared type of the subject is a class of which extensions have been defined. Thus, the actual type of subject could be one of these extensions. [FIX THIS ACCORDING TO NEW TERMINOLOGY].

In all other cases, the types of all matches must be identical.

The syntax of the type expressions in patterns is the usual type expression syntax. The type expressions are not patterns (and thus do not contain bound variables, wildcards, etc). Here is an example of pattern matching using types:

```
fun f(x:t::numeric) =>
   case x of
      y:int | y > 10 => 20.0
      z:real => z * 2.0
      (r,i):complex => r + -(i * i)
      else => 0.0
```

This is legal in Griffin because the type of x can range over the types which have a `numeric` signature.

6.3 Iterators

Iterators denote a stream of values and specify bindings obtained from this stream. Iterators appear in aggregates and in quantified boolean expressions. Iterators are also used to specify iterative actions, that is to say, loops.

iterator ::= *lhs* ∈ *domain_expression* { , *lhs* ∈ *domain_expression* } { | *condition* }
lhs ::= *name*
| *pattern*
domain_expression ::= *expression* | *arithmetic_sequence* | *iterated function*
condition ::= *boolean_expression*

The domain expression denotes a collection of values: either the components of a composite type (set, array, bag) or an explicit sequence. The iterator selects those elements of the collection that satisfy the given condition (also called the filter) and binds those elements to the lhs. If the filter is absent, then all the values in the collection are bound in succession to the lhs. Each such value becomes in turn the *current component value* of the iterator. If the lhs is a name, and the condition is true, then the current component value becomes the value of the lhs. If the lhs is a pattern, then a match is performed to determine bindings, as described in the section [Ref: Pattern matching and binding] . If the lhs is a pattern, the filter is the guard of the pattern, that is to say a component becomes the current component value only if the pattern match succeeds and the guard evaluates to true.

6.3.0.1 Examples

Let S have the value [1, 2, 3, 5, 8, 13, 21]. Then

```
x ∈ S            -- binds x to the successive values 1, 2, 3, 5, 8, 13, 21.
x ∈ S | x < 3    -- binds x to the successive values 1, 2.
x ∈ S | x > 22   -- does not bind x to any values.
```

6.3.1 Multiple Iterators

A multiple iterator denotes the direct product of several streams, controlled by a common filter. A multiple iterator binds several names simultaneously. For each current value of an outer iterator, all current values of an inner iterator are produced, and only those tuples of values that satisfy the filter are bound to the corresponding names.

6.3.1.1 Example

Let S be the same sequence as in the previous example. Then

```
x ∈ S, y ∈ S | x * y < 10   -- binds [x, y] to the successive pairs:
                             -- [1,1], [1,2], [1,3], [1,5], [1, 8],
             -- [2,2], [2,3], [2,5],
             -- [3,3].
```

6.3.2 Arithmetic Sequences

arithmetic_sequence ::= *first* [, *second*] ... *last*

Arithmetic sequences define streams of integer values. *First*, *second*, and *last* must be expressions of discrete, primitive types (integers or enumerations). If *second* is absent, an arithmetic sequence denotes an arithmetic progression with step +1 (or successor in the case of enumerations), whose values range from *first* to *last* inclusive. If *first* > *last* then the sequence is empty. If *second* is present, then the numerical difference, *D*, between consecutive elements is equal to *second* - *first*. Thus, the sequence denotes the arithmetic progression *first* + *n***D*, *n*=0,1,... If *D* > 0 and *last* >= *first*, then the last element in the series corresponds to the largest *n* such that *first* + *n***D* ≤ *last*. If *D* < 0 and *last* ≤ *first*, then the last element of the series is the smallest one such that *first* + *n***D* ≥ *last*. If *sign*(*D*) * *sign*(*last*-*first*) = -1 or if *D* = 0 then the sequence is empty.

6.3.3 Aggregates

Aggregates denote composite values of an array, record, set or bag type.

aggregate ::=
| *array_aggregate*
| *set_aggregate*
| *record_aggregate*
component_association ::=
| *expression*
| *expression* => *expression*
| *iterator* => *expression*
| *field_name* => *expression*

6.3.4 Array aggregates

An array_aggregate denotes a composite value of some array type.

array_aggregate ::= [*component_association* { , *component_association* }]

The components of the array are the values of successive expressions in each component association, from left to right. If the component association includes an iterator, then each expression selected by the iterator is part of the resulting array. Component associations that include a field_name can only be used in record aggregates.

6.3.5 Set aggregates

A set aggregate denotes a composite value of some set type. The elements of the set are the values of successive expressions in each component association, after removing duplicate values.

set_aggregate ::= { *component_association* { , *component_association* } }

6.3.6 Record aggregates

Record aggregates denote composite values of some record type.

record_aggregate ::= (*component_association* { , *component_association*})

If the component associations do not include a field name, then the components are matched positionally. Component associations that include an iterator cannot be used in record aggregates.

6.4 Labels

Any expression may be preceded by a label. Labels are used in goto statements and exit statements.

> *labelled_expression* ::= *label expression*
> *label* ::= « *identifier* »

6.5 Loops

Loops denote actions that are executed repeatedly. The action may consist of the computation of a value, where the successive values thus computed are collected into an aggregate type. In this case, the loop is a constructor. Alternatively, the actions may be performed for their side-effect.

> *loop_expression* ::=
> | *loop* | *for_loop* | *while_loop*
>
> *loop* ::= **loop** *expression_list* **end loop**
> *for_loop* ::= for *iterator loop*
>
> *while_loop* ::= **while** *expression loop*
> *exit* ::= exit [*identifier*] -- (Labeled loop identifier)

If a loop is specified with an iterator, then the action specified by the loop (the body of the loop) is executed for each value in the stream denoted by the iterator. These values are assigned to the lhs of the iterator, and to the bound variables therein. The notation {*iterator:expression*} is equivalent to set(*iterator*: *expression*); it builds a set whose elements are the successive values of expression. The notation [*iterator:expression*] is equivalent to seq(*iterator:expression*)

```
-- Need example of breaking out of loop here.
```

6.6 Quantified Expressions

Quantified expressions provide a compact notation for search loops. Existentially quantified expressions determine whether an element of a stream satisfies a given condition. Universally quantified expressions determine whether all elements in a stream satisfy a given condition. The value of a quantified expression is boolean.

> *quantified_expression* ::= *quantifier iterator*
> *quantifier* ::= ∃ | ∀

In an existentially quantified expression, the components of the stream are examined in the order specified by the iterator, until a value is found that satisfies the condition. If such a value exists, the value of the quantified expression is true and the bound variables become defined as specified by the match. If no such value exists, the expression has value false and the bound variables are undefined. The scope of these bound variables extends to the end of the enclosing construct in which the expression appears: iterator, conditional expression, while_loop, or assignment. In a universally quantified expression, if all components in the stream satisfy the condition, then the value of the quantified expression is true, otherwise the value is false. For universally quantified expressions, the scope of the corresponding bound variables extends only to the end of the iterator.

6.6.1 Examples

`x ∈ S | even(x)` -- Iterator over the even values in the composite value of S.

```
{x ∈ S | x > 0: x^2}              -- Set of the squares of the positive values in aggregate S.
{x ∈ 2..N | not ∃ y ∈ 2..x-1 | x mod y = 0: x}   -- The set of prime numbers up to N.
{x ∈ 2..N, y ∈ 2..x-1 | x mod y = 0: x}   -- The same, described with a multiple iterator.
[x ∈ S:  f(x)]                    -- Apply f to all elements of S.
[i ∈ 1..N: [j ∈ 1..N: if i = j then 1 else 0 end if]]
                                        -- N by N unit matrix, represented as an array of rows.
Btree(t ∈ file | name(t)(1) = "S": t(2..N))  -- Constructor for user-defined class Btree.
```

Let *G* denote a grammar with non-terminals *N*, terminals *T*, productions *P* and start symbol *S*. *P* is a set of productions of the form [*lhs*, *rhs*], where *lhs* is in *N* and *rhs* is an array of symbols from *N+T*. The following loop finds the symbols in *G* that can generate the empty string.

```
[N, T, P, S] = G ;
var new :={[lhs, rhs] ∈ P: lhs | #rhs = 0} ;     -- an aggregate.
var nullable := new ;
while new ≠ {}  loop                             -- a loop
    new := {[lhs, rhs] ∈ in P: lhs |
          not (lhs ∈ nullable) and               -- an aggregate.
         (∀ sym ∈ rhs | sym ∈ nullable) } ;
         -- a quantified expression
    nullable +:= new ;
end loop ;
print nullable ;
```

6.7 Basic Expressions, Operators, and Precedence

Griffin's basic expressions are for the most part identical to Ada's expressions, with two enhancement: The addition of set-algebraic operations, and combining assignment operators (as in ALGOL 68 and C. The grammar is (largely) taken from the Ada LRM.

```
basic_expression ::=
     relation { and relation }
 |   relation { and then relation }
 |   relation { or relation }
 |   relation { or else relation }
 |   relation { xor relation }
relation ::=
     simple_expression [ relational_operator simple_expression ]
 |   simple_expression [ not ] ∈ range
simple_expression ::= [ unary_addition_operator ] term { binary_operator term }
term     ::= factor { multiplication_operator factor }
factor   ::= primary [ ** primary ] | abs primary | not primary
primary ::=
     numeric_literal | void | aggregate | string_literal | name | allocator
 |   function_call | type_conversion | qualified_expression | ( expression )
qualified_expression ::= type_name ( [ expression { , expression } ] )
```

[1]

[1] In *qualified_expression*, can the constructor be empty? – JG

6.7.1 Operator Precedence

Seven levels of operators are defined, listed in increasing order of precedence:

```
assignment_operator :          ::= :=
logical_operator               ::= and | or | xor
relational_operator            ::= = | /= | < | <= | > | >=
binary_operator                ::= + | - | & | ^ | ∪ | ∈
unary_addition_operator        ::= + | - | #
multiplication_operator        ::= * | / | mod | rem
highest_precedence_operator    ::= ** | abs | not
```

Notes:

- Binary operators have *left* and *right* operands; unary operators have *right* operands. Unless otherwise stated, the evaluation order of a binary operator's operands is unspecified.
- All binary logical operators have the same precedence.
- The binary logical operators **and then** and **or else** are short-circuit forms of their corresponding full evaluation operators, **and** and **or**.
- The ∈ operator has the same precedence as the relational operators.
- Arithmetic expressions obey algebraic laws.
- The use of parenthesis is *required* in the following contexts:

Surrounding embedded assignment expressions.

Associating operands with a particular operator when successive operations encompass:

different binary logical operators.

multiple exponentiation (**), absolute value, and/or logical negation operators.

Examples:

```
b - c - d                      -- is (b - c) - d.
b - c * d / e ** f - g         -- is (b - ((c * d) / (e ** f))) - g.
b - (c := d - e)               -- embedded assignment expression - parenthesis required.
a > b and b > c and c .  d     -- adjacent "and"s - parenthesis not required.
(a > b and b > c) or c > d     -- adjacent "and" and "or" - parenthesis required.
c := a ** (i ** j)             -- adjacent "**" - parenthesis required.
```

6.7.2 Assignment Operator

The assignment operator has the lowest precedence. In the assignment expression,

```
b := 4 ** 2 + 3 * 1 <= 20 and 2 ** 3 >= 10
```

the assignment (:=) operator's right operand is false; false is assigned to b and yielded to an enclosing expression (if present). Examples:

```
b := 20                        -- assigns 20 to b;  yields 20
5 + (b := 50) / 2              -- assigns 50 to b;  yields 30
(d := (b := b / 2) - 4) / 3    -- assigns 25 to b, then 21 to d;  yields 7
```

The assignment operator is discussed in more detail in chapter [Ref: assignmentsReferences] .

6.7.3 Binary Logical Operators and Short-Circuit Control Forms

Griffin has three predefined binary logical operators: **and**, **or**, **xor** (exclusive or). Each has boolean operands and yields a boolean value.

- `b and c` is true if both `b` and `c` are true.
- `b or c` is true if at least one of `b` or `c` is true.
- `b xor c` is true if exactly one of `b` or `c` is true.

Binary logical operators have two forms: full evaluation and short-circuit evaluation. Both operands of a full evaluation operator are evaluated. The left operand of a short-circuit operator is evaluated first; if the value of the expression can be determined from the left operand's value, the the right operand is not evaluated. Full evaluation binary logical operators:

and	`b and c`	-- and; `b`, `c`: bool.
or	`b or c`	-- or; `b`, `c`: bool.
xor	`b xor c`	-- xor: `b`, `c`: bool.

Short-circuit evaluation binary logical operators:

and then	`b and then c`	-- and; equivalent to **if** `a` **then** `b` **else** `false`.
or else	`b or else c`	-- or; equivalent to **if** `a` **then** `true` **else** `b`.

Note:

- These operands may also be applied to one dimensional boolean arrays and yield a one dimensional boolean array. Both operands must have the same length and the resulting array has the same bounds as the left operand[1]:
  result [left'first + i] := left [left'first + i] *op* right [right'first + i]; i ¨ 0, 1, .. left'length - 1

Examples:

```
4 > 5 and 5 > 6                -- false.
5 > 4 and 5 > 6                -- false.
4 > 3 and 5 > 4                -- false.
4 > 5 or 5 > 4                 -- true.
5 > 4 or 6 > 5                 -- true.
4 > 5 xor 5 > 4  q             -- true.
5 > 4 xor 6 > 5                -- false.
5 > 4 or else c                -- true; c not evaluated.
5 < 4 or else c                -- c's (boolean) value
5 < 4 and then c               -- false; c not evaluated.
i /= 0 and then 10 / i > 2     -- false, if i is 0; otherwise, 10 / i > 2
(c and d) or e                 -- Parenthesis required.
(c and d) and then e           -- Parenthesis required.
```

6.7.4 Relational Operators and Membership Tests

Griffin's relational and membership test operators have a higher precedence than logical operators enabling many expressions containing both to be written without parenthesis. The relational operators compare two operands having the same (base) type; the membership operators test a value's inclusion within a range or subtype, or test whether a value of any type is a member of a set, bag, or array. These operators yield a boolean value.

=	`b = c`	-- equality; `b`, `c`: same type.

[1] Maybe this should be explicitly stated in above description. -JG

/=	b /= c	-- inequality; b, c: same type.
<	b < c	-- less than; b, c: same scalar type.
<=	b <= c	-- less than or equal to; b, c: same scalar type.
>=	b >= c	-- greater than or equal to; b, c: same scalar type.
>	b > c	-- greater than; b, c: same scalar type.
∈	b ∈ r	-- inclusion; b: scalar, r: range, subtype of b's base type.
not ∈	b not ∈ r	-- exclusion; b: scalar, r: range, subtype of b's base type.

Notes:

- "/=" and **not** ∈ are not distinct operators; they are the boolean inversions of "=" and "∈ " respectively.
- The operands of the "<", "<=", ">=", ">" operators may also be homogenenous one-dimensional arrays; the result is a boolean value, defined in terms of the operator applied to successive pairs of corresponding operand components:

b op c = true if $\forall$ i, b_i op c_i = true; false, otherwise.

Examples:

```
true /= false                -- true.
2 > 1 and true /= false      -- true;
3 ∈ 2..5                     -- true.
3 not ∈ 6..10                -- true.
```

6.7.5 Binary Operators

Griffin's binary operators are addition, subtraction, catenation, cons, set union[1]

+	b + c	-- numeric addition; b, c: same numeric type.
-	b - c	-- numeric subtraction; b, c: same numeric type.
&	b & c	-- array catenation
^	b ^ c	-- array construction (cons)[2]
∪	b ∪ c	-- union

Note:

- Mixed-mode arithmetic is prohibited. Type conversion functions may be used to transform values of one type to another type.

Examples:

```
4 - 3 - 2           -- -1
4 + 3.3             -- illegal: cannot add integer and real.
4.0 + 3.3           -- 7.3
```

6.7.6 Unary Addition Operators

The unary addition operators, +, −, #, have a higher precedence than the binary addition operators.

+	+ b	-- identity; b: numeric type.
-	- b	-- negation; b: numeric type.
#	# b	-- composite size (array, bag, map, set)

[1] This section needs additional work. Status of CONS operator to be resolved. -JG

[2] To reiterate: CONS's status will be resolved; the ^ operator is now used for pointer dereferencing.

Examples:

```
- 3                        -- -3
4 + (- 5)                  -- -1
4 + (+ 5)                  -- 9
4 - (- 5)                  -- 9
# {1, 2, 3, 5, 5, 5}       -- 6
```

6.7.7 Multiplication Operators

Griffin's multiplication operators are muliplication, division, integer remainder, set intersection and modulus:

*	`b * c`	-- numeric multiplication; b, c: same numeric type.
*	`b * c`	-- set intersection. c: same element type.
rem	`b rem c`	-- integer remainder; b, c: integer.
mod	`b mod c`	-- integer modulus: b, c: integer.

Notes:

- Integer division truncates towards zero: |*b*/*c*| * |*c*| ≤ |*b*|.
- **rem** computes the remainder when *b* / *c* is truncated towards zero: *b* **rem** *c* = *b* - (*b* / *c*) * *c*.
- **mod** computes the remainder when *b* / *c* is truncated towards - ∞:

  b **mod** *c* = *b* - *n* * *c* = [0..(c-1)], *c* > 0; [(*c*+1)..0], *c* < 0 where *n* is an integer.
- Integer division example:

b	c	b / c	b **rem** c	b **mod** c
23	7	3	2	2
23	-7	-3	2	-5
-23	7	-3	-2	5
-23	-7	3	-2	-2

Examples:

```
3 * 4                      -- 12
3 * (- 4)                  -- -12; parenthesis required.
10 / 4                     -- 2 (integer division).
10.0 / 4.0                 -- 2.5 (real division).
10.0 / 4                   -- Illegal; cannot divide real by integer.
5 mod 3                    -- 2
5 rem 3                    -- 2
```

6.7.8 Highest Precedence Operators: Exponentiation, Absolute Value, Not

Griffin has three predefined operators at the highest precedence level:

**	`b ** c`	-- exponentiation `(b^{c})`; b: numeric type, c: integer.
abs	`abs b`	-- absolute value; b: numeric type.
not	`not b`	-- logical negation; b: bool or bool array type.

Notes:

- Exponentiation: the expression `b ** c` is defined as follows:
- c is a positive integer, b is an integer or a real. b^c is b multiplied c times.

- c is zero. If b is an integer, b^c is 1. If b is a real, b^c is 1.0.
- c is a negative integer, b is an integer. Illegal; Use math library function on Float (b).
- c is a negative integer, b is a real. b^c is the reciprocal of $b^{|c|}$.
- c is a real value, b is an integer or a real. Illegal; use math library function.
- Logical negation: **not** true is false; **not** false is true.
- The operand of the **not** operator may be a one dimensional boolean array. The result is an identically-typed object, each of whose components is the boolean inversion of its corresponding operand component.

Examples:

```
2 ** 3                          -- 8 (is 2 * 2 * 2).
2 ** 0                          -- 1
2.0 ** 3                        -- 8.0
2.0 ** (-3)                     -- 0.125 (is 1.0 / (2.0 * 2.0 * 2.0)).
2 *** (-3)                      -- illegal:  integer base, negative power.
2 ** (3 ** 2)                   -- 512;  parenthesis required.
- 2 ** 4                        -- -16 (is - (2 * 2 * 2 * 2)).
(- 2) ** 4                      -- 16 (is (-2) * (-2) * (-2) * (-2)).
abs (-5)                        -- 5
abs b / c                       -- is equivalent to (abs b) / c.
not b and c                     -- is equivalent to (not b) and c.
not a and not b and not c       -- is equivalent to (not a) and (not b) and (not c).
(not a and not b) or not c      -- parenthesis required to order "and" and "or".
```

6.7.9 Combining Assignment Operators

The expression

```
b op:= expr               -- op is a binary operator;  expr is an expression is equivalent to
b := b op ( expr )
```

The operator "*op* : =" is called a combining assignment operator. Its left operand (which may contain function calls and have side-effectside-effects) is evaluated *twice*, not once. For example, if b has the value 6, then the expression

```
b +:= 4
```

assigns 10 to b (and yields the value 10 to an enclosing expression, if present). Note:

- Why parenthesize embedded assignment expressions? Consider the ALGOL 68 expression,

  ```
  b -:= c +:= d
  ```

  What is the left operand of "+ : ="?

Examples:

```
b := 3 + 4 ** (2 * 2)           -- assigns 259 to b;  yields 259
b -:= 12 * 3 + 3                -- assigns 220 to b;  yields 220
b /:= 11                        -- assigns 20 to b;  yields 20
d := 3 + (b +:= 6) / 2          -- assigns 26 to b, then 16 to d;  yields 16
(d := (b +:= 2) - 4) / 2        -- assigns 28 to b, then 24 to d;  yields 12
r := not (s := false))          -- assigns false to s, true to r;  yields true.
r and := s                      -- assigns false to r;  yields false.
```

6.7.10 User-defined Operators[1]

Predefined operators may be overloaded but their precedence and associativity are unchanged.

6.7.11 Operator Summary Table

The following table summarizes the Griffin operators discussed in this chapter. Binary infix operators have left and right operands. Unary operators have a right operand. Operators are listed in increasing order of precedence (levels 1 through 7). Within a precedence level, operators associate left to right. Notes:

- α denotes any data type.
- α_{dis} denotes a discrete data type (bool, char, int, enumerated).
- α_{num} denotes int or real.
- α_{scalar} denotes α_{dis} or real.
- **arr** [] α denotes a *one-dimensional* array whose components have type α.
- **bag** {α} denotes a bag whose components have type α.
- **map** {α} β denotes a map whose domain elements have type α and whose range elements have type β.
- $R_{\alpha_{scalar}}$ denotes a α_{scalar} range.

Table 1: Operator Summary Table

Operator	Priority	Signature	Example	Yields
:=	1	$\alpha * \alpha \rightarrow \alpha$	i := 5	5
and	2	bool * bool → bool	true and false	false
		arr [] bool * **arr** [] bool → **arr** [] bool	Componentwise and	
or	2	bool * bool → bool	true or false	true
		arr [] bool * **arr** [] bool **arr** [] bool	Componentwise or	
xor	2	bool * bool → bool	true xor false	true
		arr [] bool * **arr** [] bool → **arr** [] bool	Componentwise xor	
=	3	$\alpha * \alpha \rightarrow$ bool	'd' = 'D'	false
/=	3	$\alpha * \alpha \rightarrow$ bool	'd' /= 'D'	true
<	3	$\alpha_{scalar} * \alpha_{scalar} \rightarrow$ bool	3 < 3	false
		arr [] α_{discr} * **arr** [] $\alpha_{discr} \rightarrow$ bool	"ABC" < "AC"	true
<=	3	$\alpha_{scalar} * \alpha_{scalar} \rightarrow$ bool	3 <= 3	true
		arr [] α_{discr} * **arr** [] $\alpha_{discr} \rightarrow$ bool	"ABC" <= "AC"	true

[1] The latest decision is that all the operators can be redefined; they retain the same precedence and associativity. No new operators can be defined. This section should be updated. -- Rai.

Table 1: Operator Summary Table

Operator	Priority	Signature	Example	Yields
>=	3	$\alpha_{scalar} * \alpha_{scalar} \rightarrow$ bool	3 >= 3	true
		arr [] α_{discr} * **arr** [] $\alpha_{discr} \rightarrow$ bool	"ABC" > "AC"	false
>	3	$\alpha_{scalar} * \alpha_{scalar} \rightarrow$ bool	3 > 3	false
		arr [] α_{discr} * **arr** [] $\alpha_{discr} \rightarrow$ bool	"ABC" >= "AC"	false
Œ	3	$\alpha_{scalar} * R_{\alpha_{scalar}} \rightarrow$ bool	3 0 .. 10	true
		α * **bag** $\{\alpha\} \rightarrow$ bool	3 {1, 2, 3, 6}	true
not ∈	3	$\alpha_{scalar} * R_{\alpha_{scalar}} \rightarrow$ bool	3 not 0 .. 10	false
		α * **bag** $\{\alpha\} \rightarrow$ bool	3 not {1, 3, 6}	false
+	4	$\alpha_{num} * \alpha_{num} \rightarrow \alpha_{num}$	2 + 3	5
-	4	$\alpha_{num} * \alpha_{num} \rightarrow \alpha_{num}$	2 - 3	-1
&	4	**arr** [] α * **arr** [] $\alpha \rightarrow$ **arr** [] α	"AB" & "CD"	"ABCD"
		arr [] $\alpha * \alpha \rightarrow$ **arr** [] α	"AB" & 'C'	"ABC"
		α * **arr** [] $\alpha \rightarrow$ **arr** [] α	'A' & "BC"	"ABC"
		$\alpha * \alpha \rightarrow$ **arr** [] α	'a' & 'B'	"AB"
»	4	**bag** $\{\alpha\}$ * **bag** $\{\alpha\} \rightarrow$ **bag** $\{\alpha\}$	{1} ∪ {1, 5}	{1, 1, 5}
+	5	$\alpha_{num} \rightarrow \alpha_{num}$	+ 2	2
-	5	$\alpha_{num} \rightarrow \alpha_{num}$	- 2	-2
#	5	**bag** $\{\alpha\} \rightarrow$ int	# {1, 2, 2}	3
		arr [] $\alpha \rightarrow$ int	# [1, 2, 3, -2]	4
		map $\{\alpha\}$ $\beta \rightarrow$ int	# {3 => 2}	1
*	6	$\alpha_{num} * \alpha_{num} \rightarrow \alpha_{num}$	2 * 3	6
/	6	$\alpha_{num} * \alpha_{num} \rightarrow \alpha_{num}$	7.5 / 3.0	2.5
mod	6	int * int → int	5 mod 3	2
rem	6	int * int → int	5 rem 3	2
**	7	int * int (≥ 0) → int	2 ** 3	8
		real * int → real	2.2 ** 3	10.648
abs	7	$\alpha_{num} \rightarrow \alpha_{num}$	abs (-3)	3
not	7	bool → bool	not true	false

Table 1: Operator Summary Table

Operator	Priority	Signature	Example	Yields
		arr [] bool → arr [] bool	Componentwise not	

6.8 Undefined Values

Variables that have been declared but not yet assigned a value are **undefined**. The **defined?** operator may be applied to a name; it returns `false` if a name is undefined and `true` otherwise.

defined_expression ::= **defined?** *name*

Example:

```
-- Suppose x_def and y_def are boolean variables declared outside the let expression.
let
  x: real;
  y: real;
in
  y := 3.14_159;
  x_def := defined? x;   -- is false
  y_def := defined? y;   -- is true
end;
```

6.9 Composite Expressions

composite_expression ::=
        if_expression
| *case_expression*
| *let_expression*
| *compound_expression*

6.9.1 If expression

if_expression ::=
        if *condition* **then** *expression*
        { **elsif** *condition* **then** *expression* }
        [**else** *expression*]
        end if

The conditions following **if** and **elsif** are computed in sequence until one of them evaluates to `true` (or all have evaluated to `false`). If a condition yields the value `true` the corresponding expression is executed; its value is the value of the **if** expression as a whole. If none of the conditions is `true` and an else clause is present, the expression following **else** is similarly executed; if an else clause is not present, the **if** expression yields void.

6.9.1.1 Example

```
if inflation then blame_congress;
elsif recession then blame_federal_reserve;
elsif crisis then blame_foreign_markets;
elsif scandal then claim_ignorance;
else doze_off;
```

end if*;*

6.9.2 Case expression

case_expression ::=
case *expression* **of**
match
[**else** *expression*]
end case

6.9.3 Let expression

let_expression ::= **let** *declaration_list* **in** *expression* **end**

The **let** expression declares local bindings and yields the value of the expression. It is identical to the corresponding expression in functional languages.

6.9.4 Compound expression

compound_expression ::= **begin** *expression* [**where** *declaration_list*] **end**

The compound expression also declares local bindings and yields the value of the expression. The declaration list is optional.

6.10 Miscellaneous statement forms

The following constructs do not have a defined value, and their purpose is to affect the flow of execution of a program. When these constructs appear in expressions, the value of the expression as a whole is undefined. These familiar imperative constructs should seldom, if ever, appear in expression contexts.

6.10.1 Goto Statement

The **goto** statement transfers control to a labeled statement.

goto_statement ::= **goto** *identifier*

A goto statement can only transfer control to an expression or statement contained in the same expression_list or in an enclosing expression_list, and never into an enclosing or different expression_list.

6.10.2 Exit statement

The **exit** statement is used to terminate a loop. If the statement has no loop identifier, then the innermost loop execution is terminated. If a loop identifier is present, it must be the label of an enclosing loop. The labelled loop, and all inner loops to it are then terminated.

6.10.3 Return Statement

return_statement ::= **return** *expression* | **return**

The **return** statement terminates the execution of a subprogram and returns the value of the expression as the value of the subprogram call.

CHAPTER 7: Assignments and References

7.1 Assignments

Identifiers name values; they are either variables, in which case their value may change as the program executes, or they may be constants, in which case they keep their same initial values throughout their life. When an identifier is declared to be a variable, it may change a value with an assignment.

```
assignment_expression ::=
        lhs := expression
lhs     ::=
        variable
|       array_component
|       record_component
|       pattern
```

An *assignment_expression* evaluates to the expression on the right hand side of the operator; it has the side effect of changing the values of the variables in the left hand side. Assignment is right associative: in a concatenation of assignment, the order of evaluation is right to left. Assignment has value semantics for primitive types and composite types. That is if **x** and **y** are variables of a primitive or composite type **t**, after the assignment **x := y**, a further modification to the variable **y** will not affect the value of **x**. Assignment has pointer semantics for functions, procedures, tasks and channels. That is if **x** and **y** are variables of a function. procedure, task or channel **t**, after the assignment **x := y**, a further modification to the variable **y** will affect the value of **x**. An assignment, where the left hand side is an *array_component* or a *record_component*, is called a **destructive or inplace update**. Only the component of the array (record) is changed. The production rules for *pattern* are the same[1] as the ones described in the chapter on expressions with *lhs* instead of *identifier*. A pattern in the left hand side may fail to match the value of the right hand side – in which case an exception is raised.

Example

```
-- n and m are integers

n := 6
n := m
m := 10

-- a, b and c are arrays of integers

a := [ 10, 20, 30, 40, 50]
b := [ 60, 70, 80, 90, 100]
c[1..5] := a
c[6..10] := b
a := b
c[n] := m
-- f is a variable of type alpha to alpha,
-- g is a function of type integer to integer
f := fn (x) => x
f := g
```

The assignment operator (as the other operators) can be defined in a user defined abstract data type. It is defined in the same way as an ordinary function except that at the moment of the definition its name is **op:=** (for the other operators the name consists of the keyword **op** followed by the operator). The expression e_1 *op* e_2, where *op* is a user

[1] But do we want also guards?

defined operator, is equivalent to "*op*"(e_1,e_2). User defined operators maintain the same arity and the same precedence as the predefined operators. It is not possible to redefine operators on the built-in types.

7.2 References

Griffin references provide for fine control of the allocation of objects and explicit sharing of data. Variables of type **ptr** to a type `t`, provide an access to objects of type `t`, that may be allocated in a manner independent of the block structure. The function **new** `(t)` is used to allocate a new object of type `t`. The expression **new** `(t)`, where `t` is a type returns a value of **ptr** `t`. The value `null` is the null pointer, and it is predefined for any type **ptr** `t` for any type `t`. Two pointers are equal if and only if they point to the same object. Implicit dereferencing is not supported in Griffin.

```
type cell
type pcell = ptr cell
type cell = rec { element : int;  next :  pcell}
var listInt, tmpList : pcell
listInt := new(pcell)
listInt.element := thatValue
listInt.next := null
tmpList := listInt                        -- tmpList points to the same object
                                          --      as listInt
tmpList.element := otherValue             -- listInt.element is changed also
```

Pointers in Griffin do not introduce dangling reference problems, because it is not possible to explicitly deallocate an object allocated with the operator new. An exception is raised when the new operator is called and the run time system is not able to allocate new storage.

CHAPTER 8: Subprograms

Subprograms are one of the forms of *program unit*, of which programs can be composed. A subprogram is a Griffin *value*, a "first-class citizen" in the language; that is, it can be stored in a variable, returned by another subprogram, etc. There are two forms of subprograms: functions and procedures. A function call is an expression and returns a value; a procedure call is an expression that returns `void`.

8.1 Subprogram Specifications

A subprogram specification specifies the signature of a procedure or a function.

```
subprogram_specification ::=
        identifier : formal_part -> type_expr
formal_part ::=
        ( parameter_specification { ; parameter_specification } )
|       ()
|       type_expr

parameter_specification ::=
        names : mode type_expr [ => expression ]
mode    ::=
        [ in ]                  -- the default
|       in out
|       out
```

Examples

```
fib : (n : nat) -> nat                                -- declaration of the fibonacci  function;
                                                      -- the mode for the parameter n is in.
fac : (n : in nat) -> nat                             -- declaration of the factorial  function
square : (x : real) ->  real
cube : real -> real
incr : (x : in out number; inc : number => 1) -> void
                                                      -- declaration of a procedure;
                                                      -- default value for inc is 1
nand : (bool, bool) -> bool
```

The specification of a function or a procedure specifies its identifier and its *formal parameters* (if any). The specification of a function specifies also the type of the returned value. A parameter specification with several identifier is equivalent to a sequence of single parameter specifications. Each single parameter specification declares a formal parameter. If no mode is given, the mode **in** is assumed. If a parameter specification ends with an expression, the expression is the *default expression* of the formal parameter. Only if the mode of a parameter is **in** a default expression is allowed. The type of a default expression must be that of the corresponding formal parameter (see chapters on typing for more details). A formal parameter of a subprogram has one of the three following modes:

in The formal parameter is a constant and permits only reading of the value of the associated actual parameter.

in out The formal parameter is a variable and permits both reading and writing of the associated actual parameter.

out The formal parameter is a variable and permits writing of the value of the associated actual parameter.

The semantics of the above effects is the following: at the start of each call, if the mode is **in** or **in out**, the value of the actual parameter is copied into the associated formal parameter; then after normal completion of the subprogram body, if the mode is **in out** or out the value of the formal parameter is copied in the associated actual parameter. An implementation may achieve the above effects in other ways too.

The mode of the parameters is part of the type of the function, but it is not part of the overloading resolution..

Subprograms must be declared before they are used. However, when a group of two (or more) subprograms are mutually recursive, no ordering of definitions can define all subprograms in this group before their use. In such cases, one or more of the subprograms must be forward declared. For example,

```
const parse_term : (arr [int] of char) -> int;

fun parse_expression (a:in out arr [int] of char) : int =>
    begin
        n := parse_term(a);
        while peek_char(a) in opset loop
            op := parse_op(a);
            n := op(n, parse_term(a));
        end loop;
    where
        var n : int;
        var op : (int, int) -> int;
    end;

fun parse_term (a:arr [int] of char) : int =>
    begin
        if peek_char(a) = '(' then
            parse_char(a, '(');
            n := parse_expression(a);
            parse_char(a, ')');
        else
            n := parse_literal(a);
        end if;
    where
        var n : int;
    end;
```

Bothe **parse_expression** and **parse_term** call the other. The call of **parse_expression** from **parse_term** is valid, because **parse_expression** was defined before its use. In order for the calls to **parse_term** from within **parse_expression** to be valid, it needs to be forward declared as done on the first line above.

8.2 Subprogram Definitions

A subprogram definition specifies the execution of a subprogram

subprogram_definition ::=
        fun *identifier formal_part* [: *type_expr*] => *expression*
| **proc** *identifier formal_part* => *expression*
| **fun** *identifier pattern* => *expression* { || *identifier pattern* => *expression* }
| **proc** *identifier pattern* => *expression* { || *identifier pattern* => *expression* }

subprogram_expression ::=
        fn [*identifier*] *formal_part* [: *type_expr*] => *expression*
| **fn** *identifier pattern* [: *type_expr*] => *expression* { || *identifier pattern* => *expression* }
| **fn** *pattern* [: *type_expr*] => *expression* { || *pattern* => *expression* }

The rule *subprograms_definition* declares an identifier and defines a subprogram. The rule *subprograms_expression* defines a subprogram but does not declare an identifier; the scope of the identifier is limited to the body of the subprogram. All subprograms that are associated with an identifier can be called recursively. All the parameters, that are not variables, given in a pattern of a subprogram body have the **in** mode. The keyword **proc** is used when the return type of the subprogram is `void`.

Examples

```
fun square(x : real) : real => x * x
fun cube(x : real) => x * x * x
proc incr(x : in out number, inc : number => 1) => x := x + inc
fun fac(n : nat) => if n = 0 then
                          1
                    else
                          n*fac(n-1)
fun equal(x,x) => true
|| equal(x,y) => false
fn(x) => x                        -- expression: identity function
                                  -- in an expression:
fn  len([]) => 0                     -- the scope of len is local to the body
||  len(_^xs) => 1 + len(xs)
fn  ([]) => zero
||  ([_]) => one
||  (_) => many
                                  -- Simple copying garbage collector
type mem = map[address] loc       -- address is a subtype of loc;
isPointer : loc -> bool           -- address, loc and isPointer are defined elsewhere
proc gc(FromMem : in out mem; ToMem : out mem;
       roots : set[address]; free : out set[address])
        let
                var live : set[address] := roots
                proc copy(cell : loc) =>
                        ToMem(cell) := FromMem(cell)
                        if isPointer(cell) then
                                live with:= cell
                                copy(FromMem(cell))
        in
                for addr in roots loop copy(addr)
                [FromMem, ToMem] := [ToMem, FromMem]      -- swap spaces
                free := domain FromMem - live
```

Declarations local to a subprogram can be provided using the let expression. The result of a function is the value of the expression or it is the value returned by the `return` expression. See the chapter on expressions for more details. The form

fun `f` *formal_part* : *type_expr* => *expression*

which is the definition of the constant `f`, is equivalent to the form

`f` = **fun** *formal_part* : *type_expr* => *expression*

8.3 Subprogram invocation

A subprogram call is either a function application or a procedure call; it invokes the execution of the corresponding subprogram body.

subprogram_call ::=

```
        subprogram_name actual_parameter_part
actual_parameter_part ::=
        ( parameter_association { , parameter_association } )
parameter_association ::=
        -- empty
|       [ formal_parameter => ] actual_parameter
actual_parameter ::=
        expression
|       _
```

A parameter association is said to be *named* if the formal parameter is named explicitly; otherwise it is said to be *positional*. For a positional association, the actual parameter corresponds to the formal with the same position in the formal part. Named association can be given in any order. An actual parameter associated with a formal parameter of mode **in out** or **out** must be a variable. In a subprogram call, all the supplied parameters must use the same parameter association, either named or positional. Regardless of which parameter association is used, a subprogram call may specify a subset of the formal parameters. If the parameters not included in the call do not have a default expression, then the subprogram call yields a new subprogram; its type is described in the chapter on typing. A subprogram call which yelds a new subprogram is called a partial application. The default expression of a parameter can be overwritten by another expression, or it can be discard in order to obtain a partial application. The character _ (underscore) is used to discard a default expression without supply another expression. If positional association is used in a subprogram call *c* to *f* which yields a new subprogram *f'*; the position of the parameters in *f'* correspond to the position of the paramters in *f* where all the supplied parameters in the subprogram call *c* are moved to the leftside of the list. In a partial application, where named parameter association is used, the name of the function must be the identifier of a function and cannot be a variable. The parameters are evaluated before the call. The order of evaluation of the parameters is not specified by the language. Similarly, the order in which the values of **in out** or **out** parameters are copied back into the corresponding actual parameters is not specified by the language.

Examples¢;

```
square(z)                                -- return a value
fac(BigNumber)                           -- return a value
                                         -- Many different ways to call the procedure incr:
incr(time, quantum)                      -- return a value;
                                         -- positional parameter association is used.
incr(inc => radius, x => distance)       -- return a value;
                                         -- named parameter association is used, square(z)
                                         -- return a value
fac(BigNumber)                           -- return a value
                                         -- Many different ways to call the procedure incr:
incr(time, quantum)                      -- return a value;
                                         -- positional parameter association is used.
incr(inc => radius, x => distance)       -- return a value;
                                         -- named parameter association is used,
                                         -- the order of the parameter is not significant.
incr(inc => step)                        -- partial specification, return a function;
                                         -- named parameter association is used.
incr()                                   -- partial specification, return a function;
                                         -- the default value of the parameter inc is used.
incr(,DoubleStep)                        -- partial specification, return a function;
                                         -- positional parameter association is used.
incr(distance, _)                        -- partial specification, return a function;
                                         -- positional parameter association is used;
                                         -- the default value of the parameter inc is not used.
incr(x => volume, inc => _)              -- partial specification, return a function;
                                         -- named parameter association is used;
```

-- the default value of parameter `inc` is not used.

CHAPTER 9: Persistent Repositories: Namespaces

A namespace is a repository for long-lived objects. A long-lived object is one whose lifetime extends beyond the termination of the program that created it. Namespac s share the visibility rules that apply to other environments (in particular *with* and *use*) as stated in the visibility chapter.

In current operating systems, long-lived objects are normally files or records. However, in Griffin any language object can be persistent. That is, persistence is orthogonal to type.

9.1 Structure

A namespace is structured into a tree of directories and subdirectories (as in the UNIX[1]-based operating systems). The set of objects in the namespace are partitioned among the directories. Objects from different directories may refer to one another, provided they are contained within the same namespace. Further, if *x* and *y* are in different directories but share a subcomponent, then that subcomponent is accessible from both objects. However, an object may not refer to an object in a different namespace. This may be a runtime-error.

Each directory is a time-varying function from identifiers to type-value pairs and from identifiers to directory names. The only constraint on the directories of a namespace is that the top-level directory must have a subdirectory called Abort. Further, Abort has subdirectories indexed by the date and time of a sequester operation. A sequester operation saves the updates of an aborted transaction (see the chapter on persistence mechanisms).

9.2 Orthogonality of Persistence

In Griffin, every language object that is either defined or declared at the top level is made persistent. For example, any data type, signature, package, function, procedure or variable at the top level is persistent.

Persistent and non-persistent variables are manipulated in the same way within a program and the data movement of persistent variables between namespaces and virtual memory is automatically performed. Furthermore, the same function or procedure can be used to manipulate both non-persistent and persistent variables if those variables are type-compatible and are passed as actual parameters.

9.3 Referencing Persistent Objects

In Griffin, different persistent objects can have the same identifier if they are in different directories or namespaces. Therefore, the handle for referencing a persistent object in a namespace consists of its namespace name, directory path and symbolic identifier.

A persistent object can be referenced within a program in two ways. One way is to import the entire directory in a namespace where the object is stored. The syntax for importing a directory in a namespace within a program is the declaration:

use *namespace.directory-path*

Namespace name and directory name must be compile-time constants. The identifiers of all persistent objects such as data types, functions and variables in the directory are made visible from the point where the use namespace decla-

[1] UNIX is a trademark of Unix System Laboratories in the United States and other countries.

ration occurs to the end of the block that immediately encloses the declaration. While the identifier of a persistent object is visible, the object can be referenced through its identifier name in the same way as non-persistent variables.

Any such visible identifier will *take its meaning* from the specified namespace-directories except wher:

1. there is a local declaration for the same identifier; or
2. another directory in a namespace is imported and the directory contains a persistent object with the same identifier.

It is also possible to reference a persistent object by using its identifier qualified with its namespace and directory. That is called a *qualified reference*. In this case, the namespace name and directory name must be compile-ume constants. Through qualified references, a persistent object in a directory of a namespace which is not imported at that point can be referenced.

Each persistent object in a namespace has a type and every reference to it within a program is statically typechecked

9.4 Creating and Destroying Persistent Objects

In Griffin, declarations of data types and signatures as well as definitions of variables, functions, procedures and packages can be compilation units. When either a declaration or a definition is a compilation unit, it is said to be at top level. In Griffin, when either a declaration or a definition at top level is compiled, the object being declared or defined is created in the directory of the namespace which is being imported at that point. That is, an object which is declared or defined at top level is made persistent and created in a namespace at compile time.

If no namespace-directory is being imported at that point, those persistent objects are created in the user's default namespace-directory.

There are two ways to destroy persistent objects. One way is to redefine or redeclare them. The compilation of those declarations or definitions results in the destruction of the old persistent objects and the creation of new ones with the same names. However, in this case, the compiler asks the user for confirmation that new objects replace the old ones. If the user confirms it, then the old ones are destroyed and the new ones are created. Then, the programs which were compiled to use the old ones must be recompiled.

The other way is to use an extra-linguistic operation to destroy them. In this case, the Prototech Environment operations which will be explained in the next section are used.

9.5 Management of Namespaces

Since long-lived objects are stored in namespaces and a number of users are likely to work on namespaces, extra-linguistic operations are necessary for the management of namespaces. For that purpose, the Persistent Object Store and the Prototech Environment are provided at the operating system level.

In addition, there are the Griffin administrators who manage the Persistent Object Store and the Prototech Environment.

9.5.1 Persistent Object Store

The Persistent Object Store is an object server to manage all namespaces that are used in Griffin application programs. The Persistent Object Store may be distributed and will eventually allow namespaces to be replicated.

I nplementation Note:

The Griffin compiler and appation programs communicate with the Persistent Object Store by means of UNIX sockets. Therefore, the compiler and application programs need be informed of the internet socket address of the Persistent Object Store. For that purpose, users are required to set shell variable **PERSISTENT_OBJECT_STORE** to the internet socket address of the Persistent Object Store at the UNIX shell environment they are running programs. However, they don't have to run the compiler or application programs on the same machine as the Persistent Object Store.

End of implementation note.

9.5.2 Prototech Environment Operations

The following operations cannot be performed from within a program, but only at the Prototech Environment level. When they occur, no programs may be running in the effected namespaces.

create namespace gives back a root and subdirectory Abort.

create subdirectory creates a subdirectory of a specified parent directory in a namespace <namespace name>

copy directory copies all objects in a source directory into a destination directory. Subdirectories are not copied. Those directories need not be in the same namespace. All objects reachable from the source directory are copied.

associate with user registers a user in the Persistent Object Store and gives him/her a default namespace. Unqualified persistent variables go in the top-level directory of the default namespace.

copy object allow an object from one directory to the other. If other objects are reachable from the object being copied, then they are also copied.

delete from namespace allows the deletion of objects from a namespace.

browse, list traverses the directories and lists the contents of a directory, respectively. That is, the environmental tool transforms identifiers to strings and values to structures conforming to the type.

protection Namespaces have ownership and access modes. The ownership and access mode of namespaces can be changed by authorized users. We use the Unix scheme of user, group and other as ownership and read-permissible and write-permissible as access mode, because it is so familiar.

A tool called **Prototech** for the above operations is provided for Griffin users and the Griffin administrators.

9.5.3 Protecting and Sharing Namespaces

Namespaces need to be protected from unauthorized access and at the same time need be shared by authorized users. For example, the standard libraries of Griffin must be readable to all users, but they must be modifiable only by the Griffin administrators. Each user must be allowed to have his/her own private namespaces where he/she can store private data and debug programs. Some namespaces also need to be accessible only to a group of users.

In Griffin, any namespace in the Persistent Object Store can be used within a program. In other words, there is no linguistic mechanism for protecting namespaces. However, the protecting and sharing of namespaces can be performed at the Prototech Environment exactly in the same way those of files in UNIX. For example, the Griffin administrators not only protect system namespaces but also allow users to share namespaces by using the ownership and access mode of namespaces. Likewise, users protect their private namespaces from each other.

The Griffin compiler checks whether use-namespace declarations and attempts to update namespaces within a program are authorized. If they are not authorized, then the compilation fails.

9.5.4 Debugging a Griffin Program

If a Griffin program is designed to update shared namespaces being used by other programs, then the experimental execution of the program without protecting the namespaces is very dangerous because the execution can corrupt or destroy the shared objects in the namespaces being used by other programs. For this reason, it is needed to prevent the execution of the program under debugging from impacting the other programs. In the current design, neither of the Prototech Environment or Persistent Object Store provides any special mechanism or tool for debugging Griffin application programs. However, the structuring mechanism of namespaces and the Prototech Environment Operations are designed to facilitate debugging. A recommended way of debugging a Griffin program is as follows:

1. Copy shared objects from shared or system namespaces into temporary directories in a private namespace by using Prototech Environment Operations.
2. Set the use namespace declarations to import the temporary directories in the private namespace instead of shared or system namespaces. Since we are allowed to use the same identifier to reference different objects if they are stored in different directories or namespaces, we don't have to rename any object in the program in order to use temporary directories in the private namespaces.
3. Test the program in the private namepsaces.
4. Reset the use namespace declarations to shared or system namespaces after the completion of debugging. Then recompile the program.

9.6 Execution Model

In Griffin, every "well-defined piece of program" (block, procedure, task, program) executes in a given environment. For example, a whole program executes in a particular environment that must be given, implicitly or explicitly, when a program is executed. Formally, an environment is:

- the environment where the call is made; e.g., the state of the Unix operating system
- a "default" personal Griffin namespace
- a "default" Griffin system namespace (corresponding to the run-time environment and to the standard libraries of Griffin).
- other intended namespaces that have been included explicitly.

9.7 An Example

In this section, we present a simple example of the installation and the maintenance of a software system written in Griffin in order to show how namespaces can be used.

Suppose a software vendor has developed software in Griffin to print out the employee records for companies. The definitions of the type of an employee record and a printing function are as follows:

```
type employee_rec = rec {
     name: string;
     employee_id: string;
     salary: int;
     address: string;
};
fun print_employees() : void =>
         for x in employee loop
            print(x);
         end loop;
```

9.7.1 Installation for a Company

Suppose the vendor wants to install the software for a company called foo. First, the vendor installs the Persistent Object Store and Prototech on the computer system of the company. Second, the vendor sets up necessary namespaces and their directories by means of Prototech Environment operations. It creates namespace foo and its directories schema, data and bin (these names are meant to be suggestive to a person familiar with database design).

Then, the vendor compiles the following code in order.

```
use foo.schema;
type employee_rec = rec {
     name: string;
     employee_id: string;
     salary: int;
     address : string;
};
```

The compilation of this code creates persistent type employee_rec in directory schema.

```
-- import directory schema
use foo.schema;
use foo.data;
var employee : set of employee_rec;
```

The compilation of this code creates persistent variable employee in directory data.

Then, the vendor installs printing functions in directory bin of namespace foo by compiling the following code:

```
use foo.bin;
fun print_employees() : void =>
       begin
         -- import directory data and so employee in the directory
        -- becomes visible from this point on.
         use foo.data;
         for x in employee loop
              print(x);
         end loop;
       end;
```

The compilation of this code creates function print_employees in directory bin which prints out the employee records in directory data.

9.7.2 Maintenance

Suppose the foo company wants to add a new function called employ to the software. It implements the following code:

```
fun employ(x : employee_rec) : void =>
       employee := employee with x;
```

Since it can't test the function on real data, the company creates a temporary directory test in namespace foo and copies the persistent data in directory data to directory test by using Prototech Environment operations. Then, it compiles the following code:

```
use foo.schema;
use foo.test; -- contains a copy of variable employee in directory data.
fun print_employees() : void =>
   for x in employee loop -- employee in directory test.
      print(x);
   end loop;
fun employ(x : employee_rec) : void =>
   employee := employee with x;
```

The compilation creates functions employ and print_employees in directory test. As a result, directory test contains variable employee, functions print_employees and employ. Then, the company tests function employ by executing the following code:

```
use foo.schema;
use foo.test;
print_employees(); -- print all employees.
-- creates a dummy employee for testing.
var dummy : employee_rec := ("Smith", "999999", 30, "??");
-- add it to the employee variable.
employ(dummy);
-- print the result to check if employ works well.
print_employees();
```

If function employ works correctly, then the company installs the function in directory bin by compiling the following code:

```
use foo.schema;
use foo.bin;
fun employ(x : employee_rec) : void =>
        begin
            use foo.data;
            employee := employee with x;
        end;
```

The test directory can now be destroyed by using Prototech Environment operations.

9.8 Rationale

1. Why are many operations at the Prototech Environment rather than the language level?

   Making identifiers first-class objects in the language introduces too many complexities and precludes the possibility of static analysis. Therefore, identifier *i* is introduced as the result of the compilation of programs in which *i* is global. However, since programs don't delete identifiers, the program has no way to delete *i*.

   Traversing and listing should occur at the environment level for the same reason. Setting ownership is naturally an environmental operation.

2. Why do we ask users to explicitly copy shared objects in shared or system namespaces during debugging instead of providing a special debugging mechanism such as versioning persistent objects ?

   Most Griffin programmers will be familiar with UNIX-style file management. Therefore, the structuring and protection mechanisms of namespaces are designed to be similar to those of the UNIX file system. In working on shared persistent data, i.e. files, most programmers are also more familiar with a copying protocol than a versioning one, though a versioning protocol could be built on top of our copying mechanism.

   The tree structure of namespaces makes it convenient to copy directories. In Griffin, the use namespace declaration also makes it possible to execute a program on temporary directories without renaming any objects, because the same identifier can be used for different persistent objects if they are stored in different directories or namespaces.

3. What is the difference between packages and namespaces ?

   Packages aim at modularizing a single program, but namespaces are intended for communication either between multiple executions of a single program or between multiple programs. For example, even though different programs are designed to use the same variable defined in a package, they will use different instances of the variable at runtime. On the other hand, if programs are designed to use the same variable in a namespace, then they will, in fact, share it at runtime.

CHAPTER 10: Visibility[1]

Declarations associate identifiers with entities such as variables, constant values, types, exceptions, implementations, specifications, directories, or namespaces. (See chapters [Ref: types] , [Ref: typesetsClasses] , [Ref: subprograms] , [Ref: namespaces] , [Ref: exceptions] for specific forms of declaration.) An identifier that is declared in a declaration is called a *defining occurrence* of that identifier. Other occurrences of identifiers are called *applied*. Every applied occurrence refers to a unique defining occurrence. A declaration has an associated *scope* of program text within which the declaration is potentially *visible*. This chapter contains the rules that specify the *visibility* of declarations and the resolution of applied occurrences of identifiers to their (unique) defining occurrence.

10.1 Principles of visibility control

In the basic execution model of Griffin every program part executes in an *environment*. An environment is a collection of bindings associating zero, one or more entities with an identifier. The rules for control of visibility rest on the basic principles of

- uniformity;
- simplicity;
- safety;
- support for prototyping activity.

Uniformity is realized by using only one set of visibility rules independent of the class of declarations they refer to; in particular, the rules for resolving applied identifier occurrences for values, variables, types, packages, etc., are the same. Simplicity is accomplished by adhering to few, but established visibility principles: Algol-style nested scoping and Ada-style overloading. Sa^f^ety is addressed by insisting that possible ambiguities be resolved explicitly by the programmer; in particular, if an applied occurrence can be resolved in more than one way an error is signaled. Support for prototyping activity finds its expression in a concise mechanism for introducing collections of identifiers into the current environment without a need to explicitly name them all; specifically, the "use"-clause of Ada (or the "with"-statement of Pascal) is used with the additional possibility of renaming and exclusion of identifiers.

10.2 Basic notions

The basic notions relating to visibility control are

- *identifier and (identifier) occurrence*;
- *environment*;
- *scope*;
- *declaration*;
- *visibility*;
- *overloading and overload resolution.*

The rules of this chapter, which govern the interrelation of these notions, are collectively referred to as *visibility control*. The declarative mechanisms described in Section [Ref: visibility-control] are *visibility control declarations*. An *identifier* is a member of a predefined class of symbols. For the purposes of this chapter this class also includes *operator symbols* such as +, -, =, etc. An *(identifier) occurrence* of an identifier denotes the specific place within a

[1] This chapter is very rough. It also needs to be coordinated with the persistence chapter.

program text where that identifier occurs. An *environment* is a set of *bindings*, each consisting of an identifier and an associated entitity. A *scope* is a well-defined piece of program text such as a block, procedure body, package body, *etc*.. Scopes may be nested. A *declaration* is a linguistic mechanism for introducing a set of bindings into an environment. A *direct declaration* consists of a *definining occurrence* of an identifier, a type specification and a denotation for the bound entity. Both the type specification and the denotation may be optional. Every declaration has an associated scope. A defining occurrence is *visible* in a scope if its binding can be used in that scope according to the rules of the following sections. An identifier is *overloaded* in a given scope if there are at least two defining occurrences of that identifier visible in that scope. Associating a defining identifier occurrence with an applied occurrence in a scope is called *(identifier) resolution*. If the identifier is overloaded in that scope it is called *overload resolution*.

10.3 Visibility control declarations

There are two declarative mechanisms for controling visibility of defining identifier occurrences:

- **with** and **without** for including and excluding, respectively, specified bindings from the environment;
- **use** for entering bindings of a record or package into the containing scope.

The keywords **with** and **use** can be combined to provide a full-fledged visibility control facility with selection and renaming of identifiers.

10.3.1 Syntax

The syntax of a with declaration is:[1]

```
declaration ::-
        with identifiers
    |   with identifier : specifier as identifier2
    |   without identifier : specifier
    |   use identifier  :  specifiers
```

10.3.2 "with" declarations

An example of this is:

```
let                 -- begin surrounding scope
    declarations
    with x, y
    more declarations
in
    code
end                 -- end of surrounding scope
```

The visibility of x and y extends from the **with** statement to the end of the surrounding scope, i. e., the code denoted by *code*. This corresponds to an inner scope, which is given implicitly to avoid inconvenient deep nesting.

The type specification part, *specifier*, may be omitted if it is clear from the context. The scope of *identifier* is *code*. This is a declaration that makes *ntifier* available in *code*. There must be exactly one binding for *identifier* in the environment of this declaration that satisfies *specifier*; if there is more than one, a static error is signaled. Griffin has a block-structured visibility discipline; this means that *identifier* is visible in *code* without an explicit with declaration if it is in the static (program) environment of *code*. In this case a "with" declaration is not necessary; it serves the purpose of explicit documentation of use of global bindings and of generating an error if *identifier* is ambiguously bound in the environment. If *identifier* is not contained in the static environment, a with declaration is necessary to make *identifier* available in *code* from the dynamic environment in which *code* is to be executed. If the dynamic environment does not

[1] Need to fill out this syntax -- MCH

contain *identifier* with *specifier* then a dynamic error is signaled. Bindings can be renamed with a modified **with** declaration:

```
with identifier: specifier as identifier2
```

Its effect is that, in *code*, *identifier2* is bound to the entity associated with *identifier* in the environment of this declaration, and *identifier* is not visible in *code*. To exclude a binding from an environment a without declaration may be used.

```
without identifier: specifier
```

If the environment of this declaration contains a binding for *identifier* satisfying type specification *specifier* then it will not be visible in *code*. If no such identifier exists it has no effect.

10.3.3 "use" declarations

The basic form of a use declaration is:

```
-- begin of surrounding scope
declarations
use identifier: specifier, ...
code
-- end of surrounding scope
```

The type specification, *specifier*, is optional. The scope of *identifier* is *code*. A *multiple use* declaration consists of multiple *identifier-specifier* pairs after the use keyword. If *identifier* is bound to an entity containing bindings, such as records or packages, this use declaration has the effect of adding all these bindings to the environment of *code*. The *specifier* part is optional. Renaming of components is accomplished by modifying a use declaration with a with declaration.

```
use identifier: specifier
    with id1: spec1 as id1',
        id2: ospec2 as id2',
      identifier2: specifier
          with ...
```

The type specifiers are optional. selective use of components can be specified in combination with a **without** declaration

```
use identifier: specifier
    without id1: spec1
```

or an **only** declaration.

```
use identifier: specifier
    only id1: spec1, ...
```

On the one hand, binding for an identifier brought into scope by a use declaration hides all bindings previously visible for the that identifier (vertical hiding). On the other hand, all bindings brought into scope by a multiple use declaration are visible (horizontal visibility). [Footnote: This rule may be relaxed by adopting a rule of separate "identifier universes", sometimes also called "namespaces", but obviously not here.]

10.4 Identifier Resolution

Evaluating an identifier in an environment consists of finding a binding for that identifier in the environment and returning the associated entity. *Identifier resolution* is the first part of this process in a static environment. It consists of finding the bindings in the static environment of a scope for all the applied identifier occurrences in that scope. For every scope there must be exactly one solution to this process; if not, a *resolution error* is signaled. If the static environment is *overloaded* — i.e., there is more than one binding for at least one identifier — and there is more than one solution possible, it is also called an *overload resolution error*.

CHAPTER 11: Concurrency

11.1 Summary

Concurrency in Griffin is implemented by tasks, objects with independent threads of control. Griffin tasks are a generalization of Ada tasks; the main generalizations are:

- Tasks types, like functions and procedures, can have parameters. This allows tasks to have access to their own (externally determined) identity at the time of creation, such as their index in an array, or a printable name (their identity is available as *self*).
- The type of a task, like the type of a procedure, determines only its interface, not the code it executes. An array of tasks, for example, may contain tasks with different behaviours.
- Tasks communicate by means of channels, which are concurrent-access data-structures with multiple readers and writers. Channels provide symmetric anonymous communication between multiple servers and multiple clients. A task can send a message to a channel (akin to an entry call), and can retrieve a message from a channel, by means of an accept statement. Channels are first-class values, so that channel values can be assigned to channel variables, and passed as parameters.
- Channels may be synchronous (i.e. blocking) or asynchronous: if synchronous, sending a message will require a rendezvous with a task which is doing an accept on this channel; if asynchronous, the message will be copied into the channel, and the sender task will continue execution. Thus channels generalize the rendezvous mechanism on one hand, and message-passing systems via mailboxes on the other.

Some examples of task communication via channels are given below.

11.2 Tasks and Channels

** [1]

11.2.1 Tasks

Tasks are special forms of records whose public components are restricted to channels, and which have an independent thread of control created by a private procedure component called body. Tasks can be created in a number of ways. The simplest is by the evaluation of a em task_expression, defined as:

task_expr ::=
task { *channel_declarations* } => *expr*

The expression in the task_expression specifies the body component, which is evaluated by the separate thread of control. Accept statements in this expression can reference the specified channels. More generally, an accept statement can appear syntactically anywhere, and name any channel that is visible according to the scoping rules.

The type of a task specifies only the names and types of the channels available outside the task. Task types are specified as follows:

task_typexpr ::=
task { *channel_declarations* }

Thus, a declaration of the form:

type `tt` = **task** { *channel_declarations* };

[1] Syntax still inconsistent, being worked on -- MCH

declares a task type `tt` with named and typed channels. The identifier `tt` is then available to define constants and variables. Variables of task types are not automatically initialized so the notation:

```
var  t:tt;
```

does not create a task, only a task variable. The notation:

```
var  t:tt := task { ... } => expr;
```

can be used for initializing a task variable. This declaration is redundant, so either the task type or the channel specifications can be eliminated:

```
var t:= task { ... } => expr;
var t:tt := task {} => expr;
```

Constant task declaration can be done similarly:

```
t = task { ... } => expr;
t:tt = task {} => expr;
```

both of which initiate a task which is the value of the constant `t`.

A task type identifier can be used as a creator of a task, with the parameter being the thread-creating expression. Thus the expression:

```
tt ( body => expr )
```

will initialize the execution of a task with the specified expression as body. Note that this notation includes an implicit coercion of the expression to a procedure. This allows:

```
var t := tt ( body => expr );
```

as the notation for specifying and initializing a task. The body of a task may also be specified in the task type-expression itself:

task_typexpr ::=
        task { *channel_declarations body_declaration* }

This gives a default value of the **body** component, which can be overridden on task creation.

The channel components declared by a task are normally initialized at the time of task creation. However, they can also be initialized by supplying explicit channel values to the task creator.

11.2.2 Derived notation for tasks

For convenience, a number of derived notations are also available for task definitions. The notation:

task *ident* { *channel_declarations* } => *expr*

is allowed for declaring and initializing a constant task with an anonymous type.

11.2.3 Channels

Channels are the primary means of communication between tasks. Channels are shared queues used for message passing between threads of control. Several tasks can post messages to a channel in order to request services, and several server tasks can remove messages from channels in order to perform the corresponding service. Channels can be blocking (in which case a caller suspends until a server dequeues its request) or non-blocking (in which case a caller continues execution after posting a message, without waiting for a reply from an eventual server). A channel may be private to a task (see the section on privacy below) in which case it is analogous to the private queue attached to an Ada entry.

channel_declaration ::=

[**synch**] **channel** *identifier* : *function_typexpr* ;

11.2.4 Operations on channel types

Values of type channel provide two operations, *send* and *accept*, corresponding to sending and receiving messages over the channel. The notation for send refers to the channel as a function or procedure call, as in Ada. The accept notation is also that of Ada. Mutual exclusion is imposed on all channel operations; also, if the channel is synchronous, the sender is suspended till an accept is completed. As in Ada, the select statement is a guarded command that provides for non-deterministic accepts from several channels. As in Ada, guard expressions may not refer to the formals of the accept statement, i.e. the channel or its contents.

```
x := sfe();                            -- call (send to) sfe with value return
spe('X',12);                           -- value is void
when cnd accept sfe() => expr          -- accept statement returns value
when cnd accept spe(c,i) => expr       -- guarded dequeue
```

11.2.5 Parameterized tasks

Without adding anything more to the language, we can write a task creating function:

```
tc = fn(params) ... tt(body=>expr)...;
```

which can be used to create tasks as follows:

```
t:tt := tc(args);
```

11.2.6 Extended operations

Two extensions to the select statement allow multiple concurrent access to simulate certain reader-writer problems in a clear and efficient manner. The select statement permits the keyword `and` to indicate concurrent accepts from different channels, and the qualifier `multiple` on a single accept statement to allow many concurrent accepts from the same channel.

An ordering function can be specified to determine the order in which messages in the channel appear to the accepter. This provides flexible scheduling policies beyond the FIFO of Ada queues. (Need more detail on this).

11.3 Privacy

In general, our model allows any task which can name a task to put messages in its named channels. On the other hand, we might want a mechanism which allows a channel to be passed selectively without anyone being able to remove messages from it.

We can do this in the following way: First, we define a signature for send-only channels:

```
sig Send_Channel [ msg_type ] =
    send :  (Send_Channel [msg_type] * msg_type)   -> void
```

Then, we define a signature for read-and-send channels that is an extension of the `Sendable_Channel` signature:

```
sig Channel [ msg_type] = Send_Channel [ msg_type ] +
    read: Channel [msg_type] -> msg_type
```

In the definition of a task type, the `Send_Channel` signature would be used:

```
sig my_task = task {    ch1: ch(int) :: Send_Channel[int];
                        ch2: asynch(string) :: Send_Channel[string]
                        }
```

This would declare a task type that included two channels. The first would be a synchronous channel over which integers are sent, the second would be an asynchronous channel over which strings are sent. In both cases, the signature indicates that the channels can only be sent to from the outside world. Inside a task body, the channels would have a **Channel** signature:

```
task tt::my_task
    {
        ch1: ch(int) :: Channel[int];
        ch2: ch(string) :: Channel[string];
    } =>
    begin            - body
        ....
        accept ch1(..) ...
        ....
        accept ch2(..) ...
    end;
```

Finally, the declaration of a task would simply be:

```
foo : tt;
```

11.4 Some simple examples

11.4.1 Futures

A future can be written as a task which contains a channel which provides its value. This can be written in the form:

```
x = task
    value: ch void->int        -- a synchronous channel
    body:proc() let i = expr in loop
                    accept value() = i
```

and x.value is used to retrieve the value of the future. This requires nothing additional in the language, but we should probably allow some notation such as

```
x := *expr
```

11.4.2 Generators

Generators are simply tasks with a single value channel which is used to return successive items:

```
x = task
    value:ch void->int
    body: proc
            var l := expr
        in
            while l <> nil do
                accept value() = car(l)
                l := cdr(l)
            accept value() = undefined
..   x.value()    ...
```

In this case, a simpler version of the generator can be written as a closure:

```
x = fn()
    let var l:=expr
    in fn()
```

```
if l = nil then undefined
else projl (car(l), l:=cdr(l))
```

but this would not work if the generator state was complex.

11.5 Shared variables

Griffin tasks have access to any variables within the scope of their body. There is no guarantee that access to such variables is atomic, so it is the responsibility of the programmer to ensure this if necessary.

The Standard Prelude contains a parameterized type called **shared** which provides atomic read and write operations. It is implemented as a task with private components and a body which controls access to these components via channel operations. Some compilers may choose to optimize this type.

11.5.1 Implementing Channels

Above we described channels as primitive types. However, by extending the language a little, we can implement them. The following description is an attractive alternative that we think is worth presenting, although we currently prefer channels to be primitive. To implement channels in terms of lower-level constructs, what is required is the converse of the Ada select statement, to obtain a non-polling non-deterministic entry call to multiple tasks. Giver such an operation, we can implement a channel as a task with operations:

- send
- recv
- getreply

where both the sending task and the accepting task do calls on the channel. The body of an asynchronous channel is:

```
loop
    select
        accept send(msg) do enqueue(msg)
    or
        when queuenotempty =>
        accept recv(msg) do dequeue(msg)
```

Only channel tasks need to execute accept statements. Other tasks execute send and recv operations on these channels. In analogy with the Ada non-deterministic accept:

```
select
    accept ch1(msg1) do ...
or
    accept ch2(msg2) do ...
```

we provide a non-deterministic call operation:

```
select
    ch1.recv(msg1) do ...
or
    ch2.recv(msg2) do ...
end;
```

which will normally be implemented as non-busy-waiting.

Synchronous channels could then be defined by a body of the form:

```
loop
```

```
accept send(inmsg)
accept outmsg := recv(inmsg)
accept getreply(outmsg)
```

However, this requires a sender to wait for a previous sender to get his reply before taking his message, which might be constraining in some cases. A more general implementation would accept multiple sends and queue them, and do the communication with the receivers asynchronously also (this would probably require a message id).

In practice we would expect that channels would be implementable as simple passive objects protected by semaphores.

We might want to consider putting this in Griffin, rather than the channel itself. The advantage would be that things like the ordering function could be implemented without additional language facilities (i.e. within this quasi-Ada tasking model).

11.6 DELETED

11.6.1 Alternative notation for tasks

Rather than writing a task type in the form:

```
type tt = task {ins:aet; del:et; body:void};
```

we might want to use a notation which is more similar to that of procedures. Procedures have parameters, local declarations, and a body, and so do tasks. Objects have parameters (sizes of components, for example), and local declarations, but no body. This suggests the use of the following notation:

```
record  (p1:t1; p2:t2); declarations; begin statements end;
object  (p1:t1; p2:t2); declarations; begin statements end;
proc    (p1:t1; p2:t2); declarations; begin statements end;
task    (p1:t1; p2:t2); declarations; begin statements end;
```

The **declarations** would be for public components for records, private components and public operations for objects, local variables for procedures, and private and public components (including channels) for tasks. The **statements** would be initialization for records and objects, and code bodies for procedures and tasks. Keywords **public** and **private** would unify such notation even further.

CHAPTER 12: Persistence Mechanisms: Transactions

As explained in the chapter on Namespaces, namespaces are the repository of persistent objects – objects that live beyond the lifetime of the programs that create them. Persistence mechanisms control the modification of information in namespaces.

Transactions encapsulate modifications to persistent store. Historically, transactions have been used in database management systems where they control modifications to record data. In our case, transactions are associated with modifications to all the objects of our language – types as well as values. The goal is to achieve *atomicity*. That is,

- serializability – the concurrent execution of a set of transactions is equivalent to a serial ordering of those transactions that commit.
- failure atomicity – if a transaction commits, then its effects are reflected in persistent storage on all the objects it modified; if it aborts, then it has no effect on the objects it modified in their original locations (however, it may have an effect on a special abort directory).
- permanence of effect – any state changes produced to objects or types are recorded on storage that can survive hardware failures with high probability.

In Griffin, a *transaction statement* is a piece of code enclosed by the **transaction** and **end transaction** that may access namespaces (i.e., persistent objects). Each execution of a transaction statement is a transaction. The Griffin runtime system guarantees that modifications to those namespaces within the execution satisfy the above rules of *atomicity.*

Syntactically, a transaction statement is:

```
transaction_statement =
        transaction [ hoarding ] [ sequestering ]
                statements
                exception_handler
        end transaction
```

or an assignment statement that includes persistent identifiers. Semantically, such an assignment statement is merely shorthand for

```
transaction
        statement
end transaction
```

Thus, a transaction statement is at one scoping level. Since the statement enclosed by **transaction** and **end transaction** is a general Griffin statement, enclosed transaction statements may be at a different scoping level. The exception handler of a transaction can contain a **when abort** clause in which case aborts that occur in the transaction can be caught by the handler (See the chapter on exceptions.)

12.1 Some definitions

Persistent variables Those initially defined at the top level of a Griffin program and that reside in a specific namespace-directory.

Intended namespace-directories Those mentioned within a transaction. For example, if an assignment says N.d.x := 5, then (N,d) is the intended namespace-directory for that assignment. (N,d) is also the intended namespace-directory for x := 5, if x takes its meaning from (N,d). As discussed in the chapter on namespaces, *x* takes its meaning from (N,d) in the statement x := 5, if (N,d) is visible to the statement and no other namespace-directory containing identifier x is visible to the statement.

Transactions Between the **transaction** and **end transaction** statements, we are said to be *within a transaction.*

It is conceivable (usually because of aborts), that a program will repeatedly execute the code in a transaction block. Each different execution is a different transaction.

Commit A transaction that successfully reaches its end transaction construct without raising an abort is said to *commit*

Nested transactions If a transaction statement occurs within a transaction, that transaction statement is said to be *nested* within the surrounding transaction.

Child, parent, sibling A nested transaction is said to be the *child* of the immediately enclosing transaction which is called the child's *parent.* Any parent is also an *ancestor.* In general, if transaction T is the parent of T' and T' is an ancestor of T'', then T is an ancestor of T''. Two nested transactions with the same parent are said to be *siblings.*

Top-level-transaction A transaction that is unenclosed.

Environment of a transaction An environment of a transaction is a set of identifier-to-value mappings for persistent variables in namespaces.

12.2 Nested Transactions and Tasks

A child transaction has its effects only on the environment of its parent transaction according to the rules of *atomicity.* That is, either all modifications or none of them apply to its parent transaction depending on whether the child transaction commits or not.

In Griffin, multiple tasks can be invoked in a single transaction. Either all of modifications made by those tasks or *none of them has effects. Thus, not only can the* control of a transaction be multi-threaded but also all modifications made in those threads are controlled to satisfy the rules of *atomicity.*

Those tasks may also execute transaction statements, which create child or descendent transactions. In this case, those child or descendent transactions run concurrently if they are in different tasks.

A transaction commits after all tasks invoked in the transaction terminate and all child transactions commit or abort. If a transaction aborts, then that causes all its child transactions to abort and all tasks invoked in the transaction to terminate.

12.3 Semantics of Transactions

12.3.1 Commit

If a transaction T which is not a top-level transaction commits, then all modifications to persistent variables made both by it and by commited child transactions within it apply to the environment of the parent of T.

When a top-level transaction commits, then all modifications both by it and by committed child transactions within it apply to intended namespace-directories.

If a set of transactions (possibly from different programs) commit, then it appears as if there were some serial execution of those transactions, as far as persistent variables are concerned.

12.3.2 Aborts

If a nested transaction is aborted, then none of its (or its descendants') modifications to persistent variables apply to the environment of its parent transaction. If a top-level transaction is aborted, then none of its (or its descendants') modifications to persistent variables apply outside of that transaction or to intended namespace-directories.

Conceptually, any assignment made to persistent variables within a transaction are made to local copies.

On abort of a transaction, the values of updated persistent variables will be their value at the time the transaction initially accessed them. Therefore, those changes made inside the transaction are discarded as far as the environment of the parent transaction is concerned, the intended namespace-directory is concerned, and the environment of its exception handler. In particular, the environment of its exception handler reflects the updates of non-persistent variables made by the aborted transaction but not of persistent ones. On abort, the value of a persistent variable will be its value at the time the transaction accessed it unless another (non-descendant transaction) changed that value.

12.3.2.1 When Can Abort Happen?

An abort can occur in the following cases:

- **Deadlocks** If an inter-transaction deadlock occurs, then the Griffin runtime system aborts at least one of the transactions concerned.
- **Program-raised abortion** Programs can be designed to abort transactions by means of the **raise abort** statement in case fatal errors are detected.
- **Spontaneous aborts** Griffin also allows spontaneous aborts (e.g. to terminate a transaction that has not progressed for a long period of time).
- **Exceptions** An exception which occurs in a transaction aborts the transaction (unless it is caught inside that transaction) and is then propagated outside the transaction.

Thus, the language gives the following guarantees regarding aborts:

1. If a transaction has no internal concurrency, suffers no spontaneous aborts or exceptions, and executes at a time in which no concurrent programs access persistent variables, then the transaction will not abort.
2. If a program contains a transaction without internal concurrency, uses hoarding, and there are no spontaneous aborts or exceptions, then the transaction will not abort.

12.3.2.2 Abortion and Exceptions

In Griffin, abortion is treated as an exception. Therefore, abortion is propagated and caught exactly in the same way as other exceptions. (See the chapter on exceptions).

Program-raised aborts, deadlocks and spontaneous aborts cause the system-defined exception **abort** at runtime. Therefore, they can be caught by the exception handler of the **when abort** clause. An exception (including **abort**) must abort a transaction if it occurs inside the transaction and must be propagated outside the transaction.

12.3.3 Sequestering

Whereas the runtime system discards local copies of persistent variables on abort, those copies can be preserved by the use of the option *sequestering* in the transaction statement.

Griffin will preserve not only the objects accessed by the aborted transaction with the sequestering option but also all objects reachable from them.

The sequestering operation puts the copies of persistent variables in the directory current-namespace.abort.current-date-and-time. These directories can later be searched at the Prototech Environment level and new programs can be written to access them.

12.3.4 Hoarding

When the hoarding option is associated with a transaction T, we obtain the guarantee that once T begins it will not suffer deadlock with respect to other transactions. However, this guarantee does not exclude the possibility that transactions enclosed by *T* will suffer from deadlock with respect to one another. For example, two transactions nested within *T*, say T_1 and T_2, may suffer deadlock. It also does not exclude the possibility that *T* will abort due to a spontaneous abort or an exception.

Application Note: The programmer should note that this is an expensive option because hoarding will cause the system to lock too much. Particular attention should be paid to the following situations:

- Small portions of large structures are accessed and the portion accessed depends on a run time parameter that is unknown when the transaction begins. Large structures may be pointer structures or large arrays.
- Because of branching in the code, only a small portion of the data items mentioned are actually accessed.

12.4 Rationale Remarks

12.4.1 Rationale for making transactions single statements

It is easier to conceive of transactions as statements and is more uniform with the rest of the language. Since transactions cause changes to persistent storage, it is particularly important that their meaning be clear, ideally at compile-time. The alternative of allowing a many-to-one or many-to-many relationship between begin transaction statements and end transaction statements, possibly at different scoping levels, introduces many complexities. For example, a program will perform unpredictably if it encounters a begin transaction statement and then either a non-matching end transaction statement or a program termination.

12.4.2 Rationale for Abort Design Decisions

- Why don't we specify that a transaction cannot abort if a transaction has no internal concurrency, suffers no spontaneous aborts, and executes at a time in which no concurrent programs access *the same persistent variables as accessed by that transaction*?

  To do that requires a detailed discussion of the implementation of concurrency control. Different implementations (e.g. different lock granularities) will give different answers. For example, if entire namespaces are locked, then two transactions may enter a deadlock even if they access different variables within the same pair of namespaces. On the other hand, if variables or parts of variables are locked, then two transactions that access the same variables in the same namespace may enter a deadlock whereas the lock-the-namespace strategy would not have caused them to do so.

  Such a specification constrains the implementation excessively and has not been found to be pragmatically useful in any existing database environment.

- Why do we sequester rather than provide the intermediate results of an aborted transaction to the program that experienced the abort?

  There is a flexibility argument that suggests that the updated values of aborted transactions be made available to the running program, but let us examine its realm of applicability. Most transactions will commit, so the flexibility argument is irrelevant to them. Most transactions that abort will be short, so keeping aborted updates is pointless, so the flexibility argument remains irrelevant. If a long transaction is aborted, then reconciling the updates of the aborted transaction with the aborting transaction normally requires human intervention. For example, in the design world, two transactions may both update a wing design. If there is an abort, then the updates were likely incompatible. Figuring out what to do will require reasoning that one would not likely embed in a program, rendering the flexibility argument irrelevant again. Therefore the sequester option keeps the aborted data available and a new transaction can reconcile the updates of the aborted and aborting transaction.

- Why do we give the hoarding option?

  In a well-designed system, most transactions should commit. A feasible solution for those that don't is simply to try them again. However, it is occasionally the case that one simply does not wish to concern oneself with the abort case at all. The hoarding option is useful for that case. Hoarding can be implemented by pre-declaration locking in a database management system. That is, one obtains all locks that could possibly be necessary when the transaction begins. This can be very expensive. For example consider an employee database and a transaction that accesses two employees. Suppose that the pair of employees that are accessed is determined only after the transaction begins. Hoarding may still require that the transaction lock the *entire* employee database in order to be sure that this transaction not conflict with any other transaction on employees.

- Why do we allow programs with aborted transactions to continue to execute?

  Simply because this can be occasionally desirable.

- Why have we adopted a nested transaction model?

  In a long transaction, it may be desirable to isolate the effect of a single abort and to recover from it in some way.

- Why do we unify abortion and exception ?

  First, if an error causes an exception in a transaction and the exception can not be handled within that transaction, then the transaction must be aborted because it can not recover from the error. Therefore, exceptions can also abort transactions.

  Second, if a transaction doesn't have an abortion handler and an abort occurs in the transaction, then the abort needs to be propagated until it is caught by the abortion handler of an enclosing transaction. Therefore, abortion should be propagated in the same way as exceptions.

12.5 An Example

In the following example program, a file is represented as a doubly linked list of lines and each line as a one dimensional unbounded character array. Some functions are assumed to be already defined and their function should be clear from their names.

```
-- one dimensional unbounded character array
-- for a single line of a file.
type unbounded_char_array = arr [int] of char;

-- forward declaration.
type line_rec;
-- pointer to a line of a file.
type line_p = ptr line_rec;

-- a file is represented as a doubly linked list of lines.
type line_rec = rec {
    var f_link, b_link : line_p;
    var line : unbounded_char_array;
};

-- record the pointer to the first line in persistent variable top_line.
-- the variable is used when the sequestered updates are accessed.
var top_line : line_p;

fun editor(edit_file : line_p) : void =>
    let
        var command : char;
```

```
   var cur,old : line_p;
in
-- all the updates are encapsulated by a top-level
-- transaction with the sequestering option.
transaction sequestering
   top_line := edit_file;
   cur := edit_file;

   command := read_command();
   while true loop
   -- the execution of each editing command is a nested transaction.
     transaction
      old := cur;
      case command of
        'f' => cur := forward_move(cur);
        'b' => cur := backward_move(cur);
        'i' => cur := insert_line(cur);
        'd' => cur := delete_line(cur);
        'r' => cur := replace(cur);
        'R' => cur := global_replace(cur);
        'e' => exit;
        else   beep_sound();
      end case;

      -- read the next command to execute.
      command := read_command();
      if command = 'u' then
         -- if it is the undo command,
         -- then just abort the current nested transaction.
         raise abort;
      end if;

    exception
   when abort =>
        -- what the last command had done is undone.
        -- read the next command to execute.
        command := read_command();
        -- since the cursor is non-persistent, it need be
        -- explicitly recovered.
        cur := old;
    end transaction;
   end loop;

   -- confirm with the user that all the updates are made permanent.
   if not confirm_save() then
     -- the user wants to abort all updates.
      raise abort;
   end if;
 exception
   when abort =>
      -- the file is restored to the state in which this
      -- function was called.
     print_message("Exit without saving the updates.");
     print_message("The updates can be found in the Abort directory.");
    when others =>
      print_message("Unexpected Exception: must abort.");
     print_message("The updates can be found in the Abort directory.");
  end transaction;
end;
```

The following points are of interest.

- This editor doesn't deal with any disk i/o or persistent variables (except for top_line). When this editor is called, a persistent variable for a file will be passed to it. If a non-persistent variable had been passed, this function would work,but the use of transactions would not make sense.
- This editor doesn't need to make a separate copy of a file when the edit begins. Instead, it restores the original file when the user wants to undo all updates by means of the transaction mechanism.
- Even though the user undoes all updates, he could find the updated file in subdirectory current-date.current-time of the Abort directory in the current namespace because the top-level transaction has the sequestering option. He can access the file through persistent variable top_line in that subdirectory. (Note that the variable is not the same one, but the sequestered copy.)
- This editor undoes only the effect of the last command by aborting the current nested transaction.

12.6 Implementation Notes

A reasonable implementation strategy for concurrency control is to use locks. We follow normal implementation in that a nested transaction T inherits the locks it needs from its enclosing transaction, thereby preventing the enclosing transaction or any of its sibling transactions from modifying certain data while T is executing. Commit or abort both result in anti-inheriting the locks back to the enclosing transaction.

To perform recovery, updates are made separately. Aborting then consists of discarding those updates. So, the roll-back doesn't invalidate any accesses made by other transactions. Committing consists of moving those updates to the copies made on behalf of the enclosing transaction.

Because we are using a locking scheme, it is possible that a long transaction may keep a lock on a file while the person who started the transaction is on vacation. To overcome this problem, we need an external means to abort a transaction. We have not yet decided on a mechanism to accomplish this, but think that it should be feasible for a user with sufficient privilege to send an abort to a transaction (that would be a spontaneous abort).

CHAPTER 13: Exceptions

In order to write programs that are reliable and fault tolerant, unexpected situations must be dealt with. One mechanism by which the programming language can aid in this endeavor is by providing an easy way to deal with such situations. Traditionally, one of several strategies are used:

- checking for pre-conditions before an operation is attempted
- checking for post-conditions after an operation is attempted
- specifying a handler routine to be inoked in case of unexpected error

Each of these approaches has deficiencies that make them inappropriate in certain situations. Checking for pre-conditions may be expensive, checking for post-conditions is too easily neglected, and the actions that a handler routine can take are usually quite limited. Exception handling is an approach that is gaining in popularity, because it is appropriate in the many cases that the others are not, and solves many of their deficiencies.

13.1 Overview

Exceptions in Griffin are based on Ada exceptions, with significant enhancements, mainly from the areas of object-oriented languages such as C++. As in Ada, the **termination model** is used. In other words, when an exception is raised and eventually handled, the code unit raising the exception is always terminated. Exceptions are actually values of some type, so they can hold arbitrary information, as in ML, are arranged in a hierarchy, and can be caught based on their type as well as their values.

13.2 Raising an exception

To raise an exception, we use the **raise** expression:

raise_stmt ::= **raise** *expression*

13.2.1 Finding a handler

When an exception is **raised**, search for a **handler** begins. (For the purposes of this chapter, the **let** expression is equivalent to the **begin/end** expression. In the interests of brevity, it should be understood that any comments in this chapter concerning one also apply to the other.) If the exception is raised as a result of the elaboration of a declaration in the declaration section of a **let** expression, that expression is terminated immediately, and the exception is considered to have been raised at the **invoker** of the block. If the exception occurs elsewhere, the search for a handler begins at the current code unit, called the **raiser**.[1] If an appropriate handler is not found, the code unit is terminated, and the process repeats at the invoker of the raiser. When a handler is found, its code is executed, and the code unit it is attached to is then terminated.

An invoker of a code unit is one of two entities. If the code unit is the statement representing the function body, its invoker is the statement block from which it was called. Otherwise, the code unit is a block of code, which is either a **let** expression, in which case the invoker is the smallest surrounding **let** expression containing the raiser. For example,

```
fun f() =>
    let
        var q, d : int;
```

[1] It should be noted that this is the Ada approach.

```
    in
        ...
        begin
            ...
            q := if d = 0 then raise divide_by_zero else 100/d end if;
            ...
        end;
        ...
    end;

f();
```

Let us assume that function `f` is being called by the last statement in the example. Then the invoker chain is:

- the **begin/end** expression enclosing the raise
- the **let** expression
- the call expression

In this case, there is no handler, which will result in program termination if the exception is raised.

13.3 Exception definitions

The standard prelude contains the signature `Exception`, which defines the minimum set of operations necessary for an exception type, namely, the **raise** operation.

```
sig Exception = {
    op raise : (mytype) -> t::Any;
};
```

The result type of the raise operation is not `void` to allow such expressions as:

```
q := if d = 0 then raise divide_by_zero else 100/d end if;
```

The types of the expressions in both branches of the **if** must be the same for the entire expression to be valid. If the result type of **raise** were `void`, the above assignment statement would be in error, whereas if its return type is a universally quantified type `t`, the **if** expression is of type `int`.

Several predefined exception types are also provided, arranged in a type hierarchy, all based on this signature.

The top level exception type is called `root_exception`. Other exception types are simply extensions of this one. For example,

```
type root_exception::Exception = {
    op raise : ...;
};
type numeric_exception = root_exception with { };
type divide_by_zero = numeric_exception with { };
```

Note the use of "**with** { }" to indicate that `numeric_exception` and `divide_by_zero` are derived from (as opposed to aliases of) root_exception and `numeric_exception`, respectively, but with no additional operations or information. Users may extend the predefined exception types as necessary to provide their own.

(TENTATIVE)It might be possible to predefine other, additional exception types for whom the raise operation is implemented differently. For example, an exception type not extended from `root_exception` could implement something similar to CLU's propagation semantics, where an exception would only be allowed to propagate one level. If it were not caught there, it would be converted to a predefined exception of type `failure` which would be propagated as far upward as necessary. This permits exceptions with different propagation behaviors to coexist in the same program.

Allowing exceptions to be values of various types allows them to be parameterized in various ways, to define additional operations that may be performed on them, and to define hierarchies of exceptions, all through the regular type extension mechanism.

13.4 Handling exceptions

In Ada, handlers can be selective about which exceptions they will catch. This capability is extended in Griffin to allow exceptions to be caught based on their type and value. This is done by using a sort of conformity clause, with pattern matching. Since exceptions are arranged in a hierarchy, a handler can be as specific or general as desired. A tentative syntax for exception handlers follows:

handler ::= *pattern* : *type_expr* [| *guard*] => *expression*

Any variables in the pattern are implicitly declared, and their scope extends from to the end of the expression. Handlers can be given after the exception keyword in a **begin/end** block or a **let** expression. For example, in a **begin/end** block, we have

```
begin
    ... normal code ...
except
    handler_1
| | handler_2
. . .
| | handler_n
end;
```

or, in the case of a **let** statement,

```
let
    ... declarations ...
in
    ... normal code ...
except
    handler_1
| | handler_2
. . .
| | handler_n
end;
```

(TENTATIVE)Handlers may be attached to statements and expressions instead of just to blocks. The syntax is similar, but the keyword **handle** is used to suffix the expression or statement the handler(s) is(are) responsible for. The type of the handler expression is required to be the same as the type of the statement or expression it is attached to.

A handler may decide to reraise the same exception that it has caught. This is done by a **raise** statement with no operands.

13.5 Declaration of exceptional return types

(DISPUTED) Routines may declare what exceptions they raise.
If an unspecified exception is raised, it is converted to an exception of predefined type `failure`. This exception is parameterized by the actual exception originally raised. The `failure` exception need not be mentioned in routine declarations, as it may be raised by any routine. If a routine does not declare what exceptions it may raise, the default is that it may raise any exception. This allows a programmer to know what exceptions can be expected from a called routine.

13.6 Exceptions out of their scope

One item of note is that since exceptions may propagate out of the scope of the raising unit, and exception types may be non-global, it is possible to propagate an exception out of the scope of its declaration, and, as it propagates further up the dynamic chain, it may later reenter its scope. The same situation can occur in Ada, and it is handled similarly. While the exception is out of its type's scope, it is, of course, not possible to specify a handler that catches only exceptions of its type specifically, but it can be caught by a handler which catches all exceptions of an ancestor type.

If such a handler does not exist, or it is caught and reraised, and it reenters its scope, it may be caught there in the usual way, ie. the exception is not altered as a result of being out of scope.

```
fun f(g:void->void) =>
    begin
        ...
        g();
        ...
    except
        _::all Exception =>
            ... cleanup ...
            raise;
    end;

let
    type exc::Exception + {} = root_exception;
    const e:exc;
    fun g1() =>
        begin
            ...
            f(g2);
            ...
        end;
    fun g2() =>
        begin
            ...
            raise e;
            ...
        end;
in
    g1();
except
    ev:exc =>
        ... cleanup ...
end;
```

In this case, we have the let statement calling `g1`, which calls `f`, which calls `g2`. Within `g2`, exception `e` is raised. There is no handler within `g2`, so `g2` is terminated, and the search continues in f. Noticse that at this point, the exception thrown is not within its scope, which is the let statement. Here, we have a handler which catches all exceptions of any type. This handler will be used, because although the type of the exception is not visible, it is in the specified class. This handler reraises the exception, sending it to `g1`, which has no handler, causing it to propagate to the let statement, which does have a handler specific to its type, where it is finally dealt with.

13.7 Other approaches

Other approaches to exception handling discussed in the literature, and implemented in such languages as Mesa, are **the resumption and signalling models.** In the resumption model, the code unit raising the exception is always resumed after the exception is handled, while in the signalling model, it is up to the handler to decide whether to terminate or resume the raiser. These models were not used in Griffin because they are significantly more complex to understand, and seem to offer little additional utility.

CHAPTER 14: Ada Interoperability

14.1 Overview

Existing Ada code may be used with Griffin programs:

- The mapping of semantics between Ada and Griffin is well-defined.
- Griffin programs may invoke Ada subprograms[1], reference and update Ada variables, and use Ada types and constants declared in Ada packages accessed through **with** clauses.

14.2 Ada and Griffin Types

Within Griffin, each referenced Ada type may be viewed as an ADT, a **Griffin-Ada** type, on which two operations are generally defined:

- Assignment (:=): t → t
  A variable of some Ada limited private type may not be assigned outside its defining package.
- Type conversion: Griffin-Ada to Ada: t ↔ *AdaType*

A Griffin-Ada value may be viewed as an encapsulated Ada value protected from direct manipulation by an **object wrapper**.

In addition, many Ada types have a natural correspondence in Griffin. For these **convertible** types, a third operation is also defined:

- Type conversion: Griffin to Griffin-Ada: *GriffinType* ↔ *t*
  GriffinAdaType → *GriffinType* is implemented using the overloaded **ada_to_griffin** function
  GriffinType → *GriffinAdaType* is implemented using the overloaded **griffin_to_ada** function.

14.3 The Griffin view of an Ada object

Objects may be transformed as they cross the Ada-Griffin boundary:

- Ada type/object constraints may not be present in the corresponding Griffin type/object (Griffin int and Ada integer).
- Ada's type representation clauses, such as **'Storage_Size**, are inaccessible in Griffin.
- Griffin complies with restrictions in the use of Ada objects. For example, an Ada limited private type variable may not be updated outside its declaring package
- Griffin and Ada have distinct address spaces. Ada heap objects may not be directly accessed by the Griffin runtime system; they are protected by an encapsulating **object wrapper** and are sent back to the Ada runtime system for processing. Other Ada objects that may not be manipulated outside their declaring packages may be protected in a similar manner.

These transformations are encapsulated in the two type conversion operations. Specifically, the Griffin-Ada ↔ Ada conversion contains the object wrapper insertion (←) / removal (→), and the Griffin ↔ Griffin-Ada conversion contains constraint checks (particularly →).

Consider the Ada subprogram **Ada_Proc** in the package **Ada_Pack**,

[1] Direct entry calls and task instantiation not supported?

```
procedure Ada_Proc (I_Form: in out Integer) is
   ...
begin
   ...
end Ada_Proc;
```

and a call to Ada_Proc within a Griffin program:

```
var i: int;
...
Ada_Pack.Ada_Proc (griffin_to_ada (i))
```

If i's value satisfies Ada's Integer constraint, it is transformed to a Griffin-Ada Integer, Ada_Proc is invoked, and the returned **out** value is transformed to a Griffin int and assigned to i; otherwise, (Griffin's) Constraint_Error is raised.

14.4 Analogous Ada and Griffin Types

Enumerated types have identical semantics in both languages. Griffin's bool type may be defined as an enumerated type:

```
type bool = enum {false, true}
```

[1]Griffin uses Ada's character set — and will adopt Ada 9x character set extensions, if any. As noted, the Griffin int type is unconstrained in contrast to Ada's constrained Integer types. Griffin's real type is a machine-dependent floating-point type (typically, 64 bit IEEE *format*, identical to a native Ada floating-point type).

Griffin array variables need not have fixed bounds. An array variable whose type is arr [int] of int need not have fixed bounds throughout its lifetime. Records are similar in both languages; Ada's variant records may be implemented using Griffin's union type. Discriminated Ada types may be implemented in Griffin using type parameters. Taken together, these enable Ada aggregate types to be mapped into equivalent Griffin types.

Griffin has a pointer type analogous to (but not convertible with) Ada's access type, though it is rarely needed:

- Griffin maps may supersede Ada access-based structures.
- Griffin supports unconstrained array variables. The Griffin string variable assignment,

  ```
  str_var := str_var & some_string_value;
  ```

  cannot be used in Ada — Constraint_Error is violated. Equivalent Ada code uses an access type:

  ```
  type Access_String is access String;
  Str_Var: Access_String;
  ...
  Str_Var := new String' (
    Str_Var.all & Some_String_Value);
  ```

- Ada access values are allocated in the Ada heap and may not be dereferenced in Girffin.
- Griffin limits Ada access values to assignment and equality operations and parameter passing; it does not permit Ada access values to be dereferenced.

An Ada limited private type variable may be declared outside its defining package, but it may not participate in assignment or be tested for equality with another object. The Griffin runtime environment supports Ada limited private types in the same manner as Ada access types are supported: a separate Griffin pointer is created to access the Ada

[1] This should this be *explicitly* defined in the language.

object and all access to the Ada object is controlled through the pointer.

14.5 Accessing Griffin Values From Ada.

Griffin packages may be referenced in an Ada program through the **with** clause. A set of Griffin to Ada type conversion functions is defined to transform convertible Ada-Griffin values[1]. Inconvertible Griffin types are viewed as limited private types. For example, Griffin bags may only be passed to subprograms and entries.

14.6 Exceptions

Exceptions propagate across the Griffin-Ada boundary. While exceptions that serve the same purpose in both languages could be treated identically, the particular set of conditions giving rise to an exception may differ. Consequently, Griffin distinguishes corresponding exceptions as it does data types. For example,

```
--  Ada package.
package Some_Ada_Package is
   function Bar (I: Integer) return Integer;
end Some_Ada_Package;

--  Griffin program.
--  Ada_Standard is Ada's Standard package.
with Ada_Standard, Some_Ada_Package;
fun foo (x: int): int =>
   begin
      if x > 0 then
         Some_Ada_Package.Bar (
            griffin_to_ada(x));
      else
         Some_Ada_Package.Bar (
            griffin_to_ada (- x));
      end if;
   except
      Constraint_Error =>
         --  Griffin exception;
         --  x invalid for Ada function.
   || Ada_Standard.Constraint_Error =>
         --  Ada's Constraint_Error raised,
         --  but not handled, within Bar.
   end;  --  foo
```

[1] Perhaps an Ada program should view a Griffin package as a Griffin program views an Ada package - with encapsulating wrappers and explicit conversion functions defined for convertible types.

CHAPTER 15: Input/Output

Interoperability enables Ada I/O packages to be freely accessed within Griffin. These packages may be used to create I/O packages for Griffin objects that do not have compatible Ada counterparts.

CHAPTER 16: The Griffin Grammar

[Footnote: We have no intention at this moment to make this grammar an LR(k) or LL(k) grammar. If we use Prolog or YACC to write the parser, we can use YACC's precedence rules to solve some ambiguity or use Prolog's DCG to parse context sensitive grammars. Program layout: I assume that the indentation is handled before parsing. Thus following grammar is similar to all free-format languages.]

16.1 Tokens

This list of tokens is not complete yet. The keyword list is copied from the file "tgrindefs" in "/cs.a/griffin/tex" and examples in previous chapters.

Keywords not in the rules:

access by domain is repeat until continue exit export import from private publiconlyto val without

Reserved Words:

aborted all as begin body case channel const data directory else end enum fn for fun hoarding if in include let loop mytype namespace new of out pkg ptr proc raise rec return self sequester sig synch task then transaction type union use var variables when while with

Predefined Identifiers:

abort abs and arr bag bool char div false int map mod not null or real rem seq set true void xor

Delimiters: (Do we have too many dots and columns?)

```
{ } (      )     [     ]     ;     ,     .     ..
...:::     :::   _     |     ||    =     /=    <
<=>>=            :=    +:=   -:=   *:=   /:=   ->
=>+-       *     /     &     **    ∪     \inn  ∃
∀quote
```

Tokens with values:

identifier integer real char string

16.2 Notation and meta symbols in the grammar

Italic font is used for nonterminals or tokens carrying values. Keywords are typed in bold face. Meta characters "::=" and "|" have their usual meanings in BNF. "[*pattern*]" represents optional part. "{ *pattern* }" represents zero or more repetitions. Quotation marks are used to escape the meaning of meta characters. All other characters represent themselves.

16.2.1 questions for Griffiners

Do we put semicolumn after each *declaration*, or only after declarations which are not ended with "}"? Is the semicolumn a terminator or a separator in the *record_declaration* and other places? Do we want both set and bag types?

Right now, this is only a merge of all production rules appeared in previous chapters, with some syntax changed. Please compare them with your newest versions and make changes when necessary.

\hfill Chih-Hung

16.3 Grammar rules

Although Griffin is an interactive programming system, we use the name *compilation_unit* to mean a *complete* expression or declaration.

16.3.1 Declarations

```
compilation_unit ::=
          environment_declaration               -- only at top level?
|         declaration
|         expression ;                              -- in the following section
environment_declaration ::=
          use namespace namespace directory directory ;
|         include as transaction variables "{" names "}" ;
-- From ''namespaces.tex''  and  ''persistence.tex''
declaration ::=
          value_declaration
|         package_declaration
|         signature_declaration
|         type_declaration
|         subprogram_specification
|         subprogram_definition
|         task_implementation
value_declaration ::=
          names : typexpr = expression ;
|         var names : typexpr [ := expression ] ;
package_declaration ::=
          pkg package_name [ formal_type_par_dec ] = [ new ] package_spec ;
|        body package_name [ formal_type_par_dec ] [ : : package_name [ actual_typexpr_par
] ]
                    = [ new ] package_body ;
signature_declaration ::=
          sig sig_name = sig_expr ;
|         sig sig_con_name formal_type_par_dec = sig_expr ;
type_declaration ::=
          type identifier = [ new ] typexpr ;
|         type identifier : : sig_expr = [ new ] adt_definition ;
|         type type_con_name formal_type_par_dec = typexpr ;
|         type type_con_name formal_type_par_dec : : sig_expr = adt_definition ;
subprogram_specification ::=
          identifier : formal_part -> type_expr
-- Type parameters are specified in the formal_part.
subprogram_definition ::=
          fun  identifier  formal_part [ : type_expr ] => expression
|       proc  identifier  formal_part => expression
|       fun identifier pattern => expression { || identifier pattern => expression }
|       proc identifier pattern => expression { || identifier pattern => expression }
task_implementation ::=
          task identifier : task_spec_name = "{" { component_definitions } "}"
```

```
sig_expr ::=
            sig_name
|           sig_con_name actual_typexpr_par
|           "{" component_declarations "}"
|           sig_name + "{" component_declarations "}"
|           sig_con_name actual_typexpr_par + "{" component_declarations "}"
|           "{" typexpr | component_declarations "}"
component_declarations ::=
            [ component_declaration ] { ; component_declaration }
component_definitions ::=
            [ component_definition ] { ; component_definition }
adt_definition ::=
|           "{" component_definitions "}"
|           type_name + "{" component_definitions "}"
|           type_con_name actual_typexpr_par + "{" component_definitions "}"
|           typexpr
formal_type_par_dec ::=
            "[" formal_type_par { ; formal_type_par } "]"
package_spec ::=
            package_name [ actual_typexpr_par ]
|           [ package_name [ actual_typexpr_par ] + ] "{" declarations "}"
actual_typexpr_par ::=
            "[" actual_typexpr { , actual_typexpr } "]"
package_body ::=
           package_spec       -- but all declarations must have implementations
formal_type_par ::=
            names : typexpr
|           names :: sig_expr
|           type_con_name formal_type_par_dec
actual_typexpr ::=
            typexpr
|           constant                  -- a constant expression
component_declaration ::=
            cv names: typexpr
|           cv fun names: function_typexpr
|           cv proc names: procedure_typexpr
component_definition ::=
            names [ : typexpr ] = expression
|           var names [ : typexpr ]  [ := expression ]
cv ::=
            [ const ]                         -- default
|           var
formal_part ::=
            ( parameter_specification { ; parameter_specificaiton } )
|         ()
|         type_expr
parameter_specification ::=
            names : mode type_expr [ :: [ all | sig_expr ] [ => expression ]
mode ::=
            [ in ]                                      -- the default
|           in out
|           out
```

16.3.2 Type Expressions

```
typexpr ::=
                int
|               real
|               char
|               bool
|               mytype
|               composite_type_declaration
|               fun ( formal_typexpr_list ) typexpr
|               proc ( formal_typexpr_list )
|               tuple_declaration
|               record_declaration
|               task_specification
|               channel_declaration
|               ptr typexpr
|               enum "{" identifier { , identifier } "}"
|               data "{" member { , member } "}"
|               union [ all ] sig_expr
|               type_name
|               type_con_name actual_typexpr_par
|               constrained_type
formal_typexpr_list ::=
                formal_typexpr { , formal_typexpr_list }
formal_typexpr ::=
                mode typexpr
member ::=                                                  -- member of union type
                identifier                                  -- nullary member
|               [ identifier of ] typexpr            -- unary member
qualification ::=
                -- empty
|               qualification package_name .
function_typexpr ::=
                formal_typexpr { * formal_typexpr } -> typexpr
|               formal_typexpr { * formal_typexpr } [ -> void ]
-- procedure_typexpr is a special case of function_typexpr
procedure_typexpr ::=
                formal_typexpr { * formal_typexpr }
composite_type_declaration ::=
                seq "[" typexpr "]"
|               arr "[" index_definition { ; index_definition } "]" typexpr
|               bag "[" typexpr "]"
|               map "[" typexpr "]" typexpr
index_definition ::=
                range
|               enumerated_type_name
enumerated_type_name := type_name

range ::=
                lb .. ub : discrete_type_name
-- ``lb'' and ``ub'' are constant expressions.
discrete_type_name ::= type_name
tuple_declaration ::=
                type_expr * type_expr { * type_expr }
```

```
record_declaration ::=
        rec "{" rec_component { ; rec_component } "}"
rec_component ::= component_definition
task_specification ::=
        task "{" [ channel_declarations ] component_definitions "}"
channel_declaration ::=
        [ synch ] channel names : function_typexpr ;
-- The following are some examples from ''concurrency.tex''.
        type itt = tt  with body = procedure;    -- for task body
        type et  =  ch void -> int;      -- synchronous
        type aet  =  asynch (char,int) ->void;  -- asynchronous
        se:et;                      -- default size is 0 messages
        ase := aet(10);             -- length of queue
-- We can see that the default type of channels is synchronous in
''concurrency.tex''.
-- However, in ''grammar.tex'' the default type is asynchronous.
        x := se();       -- call (send to) se  with value return
        se('X',12);      -- value is void
        when cnd accept ase(c,i) expr    -- guarded dequeue
       when cnd accept se()   = expr   -- accept statement returns value
-- The syntax for communication between messages.
record (p1:t1; p2:t2); <declarations>; begin <statements> end;
object (p1:t1; p2:t2); <declarations>; begin <statements> end;
proc   (p1:t1; p2:t2); <declarations>; begin <statements> end;
task   (p1:t1; p2:t2); <declarations>; begin <statements> end;
-- In ''concurrency.tex'', tasks have similar syntax to records, objects,
and procedures.
constrained_type ::=
        typexpr :: [ all ] sig_expr
names ::=
        identifier [ , names ]
package_name ::= identifier
sig_name ::= identifier
sig_con_name ::= identifier
type_name ::=
        qualification identifier
namespace ::= pathname
directory ::= pathname
-- Do we use the UNIX syntax for "pathname"?
```

16.3.3 Expressions

```
-- Should we change the word ''statement'' to ''expression'' in the
following rules?
-- Do we have nested transactions?
transaction_statement ::=
        begin transaction [ hoarding ]
                statements
        end transaction
        [ abort_exception_handler ]
abort_exception_handler ::=
        when aborted [ sequester ; ] statement
abort_statement ::=
        raise abort
```

```
subprogram_call ::=
        subprogram_name   actual_parameter_part
actual_parameter_part ::=
        ( parameter_association { , parameter_association } )
parameter_association ::=
        -- empty
|       [ formal_parameter => ] actual_parameter
actual_parameter ::=
        expression
|       _
expression ::=
        transaction_statement        -- nest transaction?
|       abort_statement
|       let_expression
|       return_expression
|       compound_expression
|       if_expression
|       bag_expression
|       case_expression
|       subprogram_expression
|       subprogram_call
|       task_expression               -- undefined
|       assignment_expression
|       basic_expression
|       tuple_expression              -- undefined
|       iteration                        -- undefined
let_expression ::=
        let declaration_list in expression
-- declaration_list is undefined.
return_expression ::=
        return expression
|       return
compound_expression ::=
        "{" expression { ; expression } "}"
if_expression ::=
        if expression then expression
        [ else if expression then expression ]
        [ else expression ]
        end if
bag_expression ::=
        "{" iteration "}"
|       "{" value { , value } "}"
-- <value> is undefined.
subprogram_expression ::=
        fn [ identifier ] formal_part [ : type_expr ] => expression
|     fn identifier pattern [ : type_expr ] => expression { || identifier pattern => expression }
|       fn pattern [ : type_expr ] => expression { || pattern => expression }
case_expression ::=
        case expression of
          pattern [ "|" guard ] => expression
          { pattern [ "|" guard ] => expression }
          [ else expression ]
        end case
guard ::=
         expression
pattern ::=

|        identifier
|        literal
|        sequence_pattern
|        bag_pattern
|        tuple_pattern
|        map_pattern
|        identifier as pattern
sequence_pattern ::=
         identifier
|        "["  "]"
|        "["  pattern  { , pattern }  "]"
|        pattern ^ sequence_pattern
|        sequence_pattern ::: sequence_pattern
bag_pattern ::=
         identifier
|        bag_pattern \cup bag_pattern
|        "{" pattern { , pattern } "}"
-- Some non-ASCII characters are used in production rules about
expression.
-- Should we define their ASCII encoding sequences?
map_pattern ::=
         identifier
|        map_pattern \cup map_pattern      --- n.d. disjoint union
|        "{" pattern -> pattern { , pattern -> pattern } "}"
record_pattern ::=
|        identifier
|        ( field_name = pattern { , field_name = pattern } )
|        ( pattern { , pattern } )
iterator ::=
        lhs \inn aggregate_expression { , lhs \inn aggregate_expression }  { "|" condition }
aggregate_expression ::=
         expression
|        arithmetic_sequence
|        iteratedfunction
arithmetic_sequence :=
         first  [ , second ] ... last
-- First, second, and last must be expressions of discrete, primitive types.
loop ::=
         constructor
|        for_loop
|        while_loop
constructor ::=
         type_name ( iterator : expression )
|        "{" iterator : expression "}"
|        "["  iterator : expression "]"

for_loop ::=
         for iterator loop statements end

while_loop ::=
         while expression loop statements end
```

```
quantified_expression ::=
        quantifier iterator
quantifier ::=
        \exists
|       \forall
assignment_expression ::=
        lhs := expression
|       lhs assignment_operator basic_expression
-- assignment_operator is not in the current
''assignmentsRefenrence.tex''.
lhs ::=
        variable
|       array_component
|       record_component
|       pattern
assignment_operator ::=
        :=
|       +:=
|       -:=
|       *:=
|       /:=
basic_expression ::=
        relation [ and relation ]
|       relation [ and then relation ]
|       relation [ or relation ]
|       relation [ or else relation ]
|       relation [ xor relation ]
relation ::=
        simple_expression [ relational_operator simple_expression ]
|       simple_expression [ not ] \inn range
simple_expression ::=
        [ unary_addition_operator ] term [ binary_operator term ]
term ::=
        factor [ multiplication_operator factor ]
factor ::=
        primary [ ** primary ]
|       abs primary
|       not primary
primary ::=
        numeric_literal
|       null
|       aggregate
|       string_literal
|       name
|       allocator
|       function_call
|       type_conversion
|       qualified_expression
|       ( expression )
logical_operator ::=
        and
|       or
|       xor
relational_operator ::=
        =
|       /=
|       <
|       <=
|       >
|       >=
binary_operator ::=
        +
|       -
|       &
|       ^
|       :::
|       \cup
unary_addition_operator ::=
        +
|       -
multiplication_operator ::=
        *
|       /
|       mod
|       rem
highest_precedence_operator ::=
        **
|       abs
|       not
```